VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

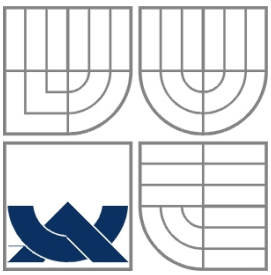
IMPLEMENTACE OBRAZOVÝCH KLASIFIKÁTORŮ V FPGA

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

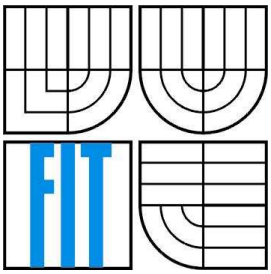
AUTOR PRÁCE
AUTHOR

Bc. FILIP KADLČEK

BRNO 2010



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

IMPLEMENTACE OBRAZOVÝCH KLASIFIKÁTORŮ V FPGA

IMPLEMENTATION OF IMAGE CLASSIFIERS IN FPGAS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. FILIP KADLČEK
AUTHOR

VEDOUCÍ PRÁCE — doc. Dr. Ing. OTTO FUČÍK
SUPERVISOR

BRNO 2010

Abstrakt

Práce je zaměřena na obrazové klasifikátory a jejich implementaci v FPGA. Klasifikátory dělí na dvě skupiny – slabé a silné klasifikátory. Ve skupině silných klasifikátorů se zaměřuje především na AdaBoost. Ve skupině slabých klasifikátorů jsou probrány základní příznakové klasifikátory, jakými jsou například klasifikátory založené na Haarových nebo Gaborových vlnkách, ale především je kladen důraz na klasifikátory LBP, LRP a LR. Naposled uvedené klasifikátory jsou vhodné pro implementaci v FGPA. Na základě těchto klasifikátorů je navržena pseudo-paralelní architektura. Architektura uvažuje provedení klasifikace v FPGA a následné zpracovávání výsledků v počítači. Navržený klasifikátor je velmi rychlý a každý hodinový cyklus produkuje výstup klasifikace.

Abstract

The thesis deals with image classifiers and their implementation using FPGA technology. There are discussed weak and strong classifiers in the work. As an example of strong classifiers, the AdaBoost algorithm is described. In the case of weak classifiers, basic types of feature classifiers are shown, including Haar and Gabor wavelets. The rest of work is primarily focused on LBP, LRP and LR classifiers, which are well suitable for efficient implementation in FPGAs. With these classifiers is designed pseudo-parallel architecture. Process of classifications is divided on software and hardware parts. The thesis deals with hardware part of classifications. The designed classifier is very fast and produces results of classification every clock cycle.

Klíčová slova

Klasifikátor, klasifikace, LBP, Local Binary Pattern, LRD, Local Rank Difference, LRP, Local Rank Patterns, Haarovy vlnky, Gaborovy vlnky, Boosting, AdaBoost, FPGA, Pseudo-paralelní architektura.

Keywords

Classifier, classification, LBP, Local Binary Pattern, LRD, Local Rank Difference, LRP, Local Rank Patterns, Haar wavelets, Gabor wavelets, Boosting, AdaBoost, FPGA, Pseudo-parallel architecture.

Citace

Kadlček Filip: Implementace Obrazových klasifikátorů v FPGA, diplomová práce, Brno, FIT VUT v Brně, 2010

Implementace obrazových klasifikátorů v FPGA

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Otty Fučíka. Další informace mi poskytli Pavel Zemčík, Roman Juránek a Antonín Hegar. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Filip Kadlček
26. 5. 2010

Poděkování

Chtěl bych poděkovat především Doc. Ing. Ottu Fučíkovi, Dr. za jeho čas stráveny při konzultacích a aktivní přístup při řešení problémů. Dále také Doc. Pavlu Zemčíkovi, Ing. Romanu Juránkovi a Antonínu Hegarovi za poskytnuté informace, spolupráci a rady při řešení práce.

Obsah

1 Úvod

Práce se zabývá především klasifikací obrazu, což je proces, při kterém se rozdělují předměty, zejména z reálného světa, do několika tříd. Základní dělení může být jen na skutečnost, zda klasifikovaný prvek patří do množiny, nebo zda do množiny nepatří. Množiny jsou předem definované třídy, které lze popsat jejich charakteristickými vlastnostmi.

Ač jsou dnešní počítače mnohem rychlejší než před deseti lety, jejich rychlost stále ještě není dostatečná pro zpracování dat v reálném čase. Proto se konstruují nové heterogenní systémy, jež se skládají z více funkčních jednotek. Vyšší výkon se u takového systému dosahuje v důsledku toho, že jednotky, ze kterých se systém skládá, jsou specializované obvody, které umí vykonávat jen určitou funkčnost. Ale danou činnost dokážou vykonávat velmi rychle a efektivně. Jsou mnohem rychlejší než univerzální multifunkční výpočetní jednotky. Takovou specializovanou jednotkou je například grafická karta v počítači. U prvních generací grafických karet nebylo možné žádným způsobem změnit jejich činnost. Jednalo se v podstatě jen o jednoúčelové zařízení. V dalších generacích grafických karet se však postupně začala prosazovat možnost programování vybraných částí grafických karet, a tím se stávají tyto karty univerzálnějšími výpočetními jednotkami, které je v dnešní době možné použít nejen pro urychlení grafických výpočtů, ale i pro obecné výpočetní úlohy.

Další možnou technologií jak urychlit vykonávání činnosti počítače je vytvořit si vlastní specializovaný ASIC[1] čip (viz kapitola 3), jež bude urychlovat vybraný algoritmus na úrovni hardwarové akcelerace. Takový čip má však využití jen v malém množství aplikací, které jsou často jednoúčelové. A tak není možné takový obvod využít v jiném druhu aplikací. Pokud dojde ke změně algoritmu výpočtu, ASIC obvod není možné změnit a musí se celý vyměnit, což je ovšem velmi nákladná činnost. Proto není tato technologie vhodná do systémů, které slouží k obecnému počítání, jímž běžný osobní počítač bezesporu je.

Mnohem perspektivněji se jeví použití technologie FPGA (viz kapitola 3). Tato technologie dovoluje nakonfigurování čipu dle konkrétních požadavků aplikace. Nakonfigurováním se zde myslí nahráním struktury obvodu, jehož funkci má FPGA čip vykonávat. Tyto obvody však ještě nejsou příliš rozšířeny v osobních počítačích. Již dnes je však možné zakoupit počítačovou kartu s FPGA čipem a připojit ji přes PCI[2] nebo PCI-expres rozhraní do běžného osobního počítače. Aplikace si poté může nakonfigurovat FPGA čip tak, aby urychloval část jejího výpočtu. Po dokončení činnosti aplikace je možné FPGA čip opětovně rekonfigurovat a spustit na něm jinou aplikaci. Tato vlastnost zavádí vysokou míru znuvupoužitelnosti FPGA čipu.

Zavedení FPGA technologie do osobních počítačů má ale několik nepřátel. Hlavním nepřítelem je vysoká cena. A to nejenom cena čipu FPGA, ale především cena návrhu aplikací (programů), jež dokážou pracovat s FPGA čipem. Technologie FPGA byla vybrána pro implementaci klasifikátoru, jež dokáže zpracovávat data v reálném čase.

Druhá kapitola práce se zabývá klasifikací obrazu. V úvodu je vysvětlen pojem klasifikace a následně je představeno několik slabých klasifikátorů. Po představení slabých klasifikátorů jsou představeny i silné klasifikátory. Z množiny slabých klasifikátorů jsou popsány například LBP, LRP a LR klasifikátory, jež jsou vhodné pro implementaci v FPGA. Z množiny silných klasifikátorů se práce zabývá například algoritmem AdaBoost. Klasifikace obrazu může například sloužit k vyhledávání osob v obraze. Práce je zaměřena především na učící se klasifikátory, které zpravidla klasifikují předměty do dvou tříd. Takový klasifikátor je výsledkem trénovacího procesu. Jelikož na

[1] ASIC - Application-specific integrated circuit – Aplikačně specifický integrovaný obvod.

[2] PCI - Peripheral Component Interconnect

trénovací proces nejsou kladeny přísné požadavky na dobu výpočtu, zpravidla se trénování provádí bez hardwarové akcelerace. Natrénovaný klasifikátor se poté sestrojí v hardwarové realizaci a může pracovat velmi rychle.

Ve třetí kapitole je popsán základní princip technologie FPGA. A jsou v ní popsány základní stavební bloky, ze kterých se FPGA skládá.

Ve čtvrté kapitole je popsáno několik architektur, které byly v rámci projektu navrženy. Některé architektury zůstaly rozpracovány jen na úrovni návrhu. Jako nejvhodnější řešení pro realizaci byla vybrána pseudo-paralelní architektura. Tato architektura není navržena jako kompletní silný klasifikátor, ale jako jednotka pro před-klasifikaci obrazu. Cílem architektury je klasifikovat co nejvíce prvků pozitivně a co nejméně vyhledávaných prvků klasifikovat negativně. Takto předzpracované prvky se následně zpracují pomocí univerzálního procesoru, nebo DSP. Cílem celého řešení je za pomocí malého a levného FPGA čipu provést předzpracování dat a následně dokončit zpracování v univerzálním procesoru.

V páté kapitole je rozebrána implementace pseudo-paralení architektury. Postupně jsou popsány všechny stavební bloky, ze kterých se skládá. Jelikož je výsledná architektura velmi proměnná, v závislosti na vstupním klasifikátoru, byl vytvořen generátor VHDL kódu, pomocí něhož se vždy získá klasifikátor pro zadané parametry. Při procesu generování jsou prováděny různé optimalizace ve VHDL kódu, které jsou popsány v této kapitole.

V šesté kapitole se práce zabývá testováním architektury. Architektura je testována jednak na rychlost, ale také na spotřebované zdroje na čipu FPGA. V závěru šesté kapitoly je uvedena realizace klasifikátoru v modulu DX64.

2 Klasifikace

S klasifikací se setkáváme v každodenním životě, aniž bychom si ji uvědomovali. Chápeme ji jako běžnou činnost v našem životě. Na světě existuje mnoho druhů aut a každé vypadá trochu jinak. Ale všechny mají kola, dveře, kapotu a další jim charakteristické rysy. Na základě těchto rysů dokážeme posoudit, zda se jedná o auto či ne. Člověk spolehlivě rozpozná auto, protože se mozek „natrénoval“ na rozpoznávání tvarů podobných autu a dokáže je klasifikovat do stejné třídy. V mozku se takové natrénování provádí spojováním perceptronů neuronů do neuronové sítě. Taková propojená neuronová síť dostane na vstup data z očí (obraz) a výstupem je zařazení obrazu do jedné z mnoha kategorií.

Mozek nepracuje s diskrétními hodnotami výstupu, ale pracuje s jistou mírou nejistoty. Mozek vždy řekne, jsem si jist, že se jedná o auto nebo mohlo by to být auto, případně asi to nebude auto. Vyjadřování mozku by se dalo přiblížit fuzzy logice[3]. Běžné klasifikátory však s fuzzy logikou nepracují. Výstup klasifikátoru může být uveden například v procentuální míře nejistoty. Klasifikátor tak může na svém výstupu poskytovat hodnotu, jsem si jist z padesáti pěti procent, nebo hodnotu nejsem si jist ze čtyřiceti pěti procent. Následně je třeba nastavit hranici, od kolika procent se bude jednat o auto. Hranice může být například nastavena na sedmdesát procent a od této hranice se vše bude považovat za auto.

Klasifikátory mohou pracovat na několika principech. Jedním z principů je, že klasifikátor se snaží v obraze najít jisté příznaky, například kola na autě, dveře a podobné charakteristické rysy klasifikovaného předmětu. Na základě těchto rysů se snaží poté rozhodnout, zda daný předmět patří do kategorie, pro kterou je klasifikátor navržen.

Doposud však ještě nebyla uvedena jedna z důležitých poznámek o klasifikaci. Mozek může klasifikovat jeden předmět paralelně do několika kategorií. Je to dáno tím, že v mozku pracuje několik klasifikátorů najednou (paralelně). Základní klasifikátor dokáže klasifikovat předměty jen do jedné třídy a o prvky dokáže říci jen to, zda do klasifikované množiny prvků patří, nebo nepatří. To však není žádnou překážkou a naopak to může být využito ve prospěch klasifikace, pokud spojíme více základních klasifikátorů do jednoho pokročilejšího klasifikátoru.

Klasifikátory můžou pracovat na základě hledání předem definované množiny příznaků nebo jsou navrženy obecně a postupem času se naučí rozpoznávat předměty. Druhý uvedený typ klasifikátoru spadá do kategorie učících se klasifikátorů. Učení může probíhat s učitelem nebo bez učitele. Klasifikátory, jež se dokážou nejprve naučit klasifikovat předměty na základě učící množiny, mohou být využity ke klasifikaci do různých množin. Ale vždy musí nejprve dojít k naučení na danou klasifikační množinu. Učení klasifikátoru je obvykle náročné a vyžaduje velkou množinu vhodných trénovacích dat.

2.1 Klasifikátor

Klasifikátor je stroj, nebo program, jenž provádí rozdělování objektů reálného světa do tříd. Většinou dokáže klasifikátor rozdělovat objekty jen do dvou tříd (jedna třída představuje klasifikovanou třídu a druhá třída je vše ostatní). Pokud si představíme klasifikátor jako stroj, může to být například kontrolní prvek na výrobním pásu, který bude výrobky, jež se na něm pohybují klasifikovat na vadné nebo správné. Například při výrobě kostek, jež mají splňovat daný rozměr, dojde na výrobní lince

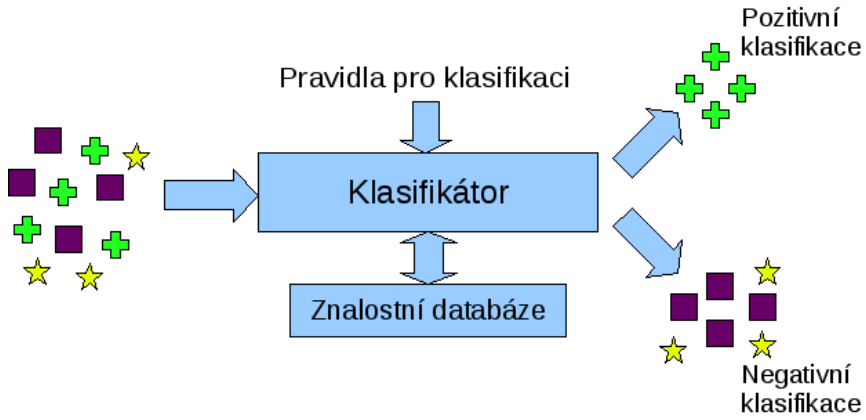
[3] Fuzzy logika – logika, která pracuje s tzv. mlhavými pojmy (fuzziness = mlhavost), umožňuje částečnou příslušnost prvku k množině (pomocí funkce příslušnosti).

k změření jejích rozměrů a dle výsledků měření se klasifikuje kostka jako vyhovující nebo nevyhovující. Nevyhovující kostka bude zřejmě z pásu odebrána a vyřazena z dalšího zpracování.

V předešlém odstavci byl uveden jednoduchý fyzicky realizovatelný klasifikátor. Z pohledu této práce jsou však takové klasifikátory nezajímavé. V práci se budu zabývat především softwarovými klasifikátory. Softwarový klasifikátor pracuje s daty, jež lze uložit do paměti počítače. Data jsou reprezentací předmětu z reálného světa. Například zvuk nás obklopuje, kdekoliv se jen dostaneme. Jelikož je běžnou součástí našeho života, byly vyvinuty techniky, jak zvuk efektivně ukládat v počítačové technice. Další velmi důležitou věcí v životě je obraz. Člověk jej vnímá pomocí očí a slouží člověku ke každodenním činnostem. Bez obrazu by byl život mnohem komplikovanější. Stejně jako u zvuku byla i u obrazu vyvinuta technologie pro jeho efektivní ukládání v dnešních počítačích. Tato práce se zaměřuje právě na zpracování rastrového obrazu pomocí klasifikátorů. Rastrový obraz představuje typickou možnost, jak ukládat obrazová data v počítači.

Obrázek 1 ukazuje základní schéma obecného klasifikátoru. Klasifikátor na obrázku rozděluje (klasifikuje) vstupní objekty do dvou tříd. Klasifikátor je naučen tak, aby správně klasifikoval zelené křížky a zařadil je do správné třídy (na obrázku naznačena textem *Pozitivní klasifikace*). Všechny ostatní objekty ze vstupu klasifikuje do třídy nevyhovujících objektů (na obrázku naznačeno textem *Negativní klasifikace*). Klasifikátor pro svou činnost využívá pravidla ke klasifikaci, dle nichž řídí klasifikaci. Pokročilejší (učící) klasifikátory mohou během své činnosti vytvářet znalostní databázi a během své činnosti z ní čerpat. Někdy je ovšem vhodné rozdělit proces ukládání dat do znalostní databáze od procesu, kdy se znalostní databáze využívá pouze ke klasifikaci a nezasahuje se do ní. První fázi se často říká učení klasifikátoru.

Z obrázku 1 lze vidět, že klasifikace proběhla správně a klasifikátor správně rozdělil objekty ze vstupní množiny do dvou tříd. V první třídě jsou zelené křížky a v druhé třídě jsou fialové čtverečky a žluté hvězdy.



Obrázek 1: Klasifikátor

2.2 Vektor příznaků

Klasifikátory obvykle zpracovávají data z reálného světa. Jelikož prozatím není možné pomocí současné počítačové techniky reprezentovat objekty reálného světa se všemi detaily v paměti počítače, musí se provést převod objektu z reálného světa na tzv. vektor příznaků. Vektor příznaků je reprezentace objektu reálného světa v paměti počítače.

Pod vektorem příznaků se může skrývat i rastrový obraz. Dívá-li se člověk na nějaký předmět, může se zaměřit na jeho různé detaily nebo na předmět jako celek. V počítačové paměti

však můžeme uložit jen celkový obraz. Obraz však můžeme uložit s určitým rozlišením. Právě toto rozlišení nám říká, jak mnoho detailů jsme uložili. Uložení obrazu, jež vidíme lidským okem, do paměti můžeme nazývat extrakce příznaků. V tomto případě ukládáme spojitou veličinu, jíž je vstupní obraz (nekonečná úroveň rozlišení – proto spojitá), jako diskrétní hodnoty (omezené rozlišení). Jak již bylo dříve napsáno, v počítačích jsme schopni uložit obraz jen s omezeným rozlišením, a proto musíme provést výběr příznaků z reálného světa. Příznak, který ukládáme u rastrového obrazu, je jeho barevná reprezentace. Barvu ukládáme ke každému bodu obrazu (pixelu) zvlášť. Použijeme-li RGB[4] model, ukládáme ke každému bodu obrazu 3 hodnoty, z nichž každá vyjadřuje jednu základní barvu obrazu. Takový uložený obraz můžeme nazývat vektor příznaků - *feature vector*.

Získaný vektor příznaků však dále ještě zpracováváme a snažíme se získat relevantní informace pro aplikaci, ve které se bude vektor zpracovávat. Tento postup se v anglické literatuře často označuje jako *feature extraction*. Hlavní funkcí uvedeného mechanizmu je vybrání důležitých příznaků a tak zmenšení objemu dat vstupujícího do klasifikace.

V případě klasifikace obrazu může být například zbytečné uchovávat barevný obraz, jelikož klasifikace může stejně dobře probíhat i na obraze s odstíny šedi. Extrakce příznaků bude tedy spočívat v tom, že se převede barevný obraz na reprezentaci s odstíny šedi. Tím se výrazně zmenší objem dat, jež je nutné zpracovávat (objem dat se zmenší o dvě-třetiny).

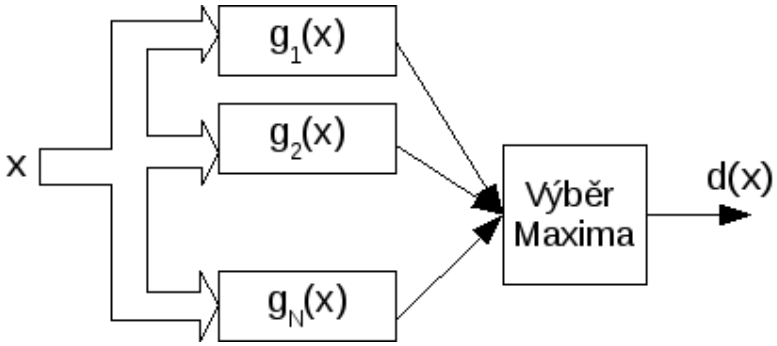
Dalším příkladem získávání příznaků může být zpracovávání vyplněného testu. Z vyplněného testu se vyberou jen uživatelem zvolené odpovědi na otázky a vytvoří se tak vektor příznaků. Tento vektor příznaků může být dále zpracován. Extrakce příznaků v tomto případě spočívá jev v převodu vyplněného testu - vybraných odpovědí, na číselnou reprezentaci, která může být v počítačích lehce zpracována.

2.3 Význam učení v klasifikaci

Klasifikátory mohou být implementovány buď jen na jeden konkrétní typ klasifikace, nebo na různé druhy klasifikace, které se často naučí až v rámci svého běhu. Prvním druhem klasifikátorů jsou klasifikátory bez možnosti učení. Tyto klasifikátory mají danou přesnou množinu pravidel, dle nichž vyhodnocují své vstupy. Pravidla bývají obvykle napevno zapsána do kódu klasifikátoru. Vstupem takového klasifikátoru je obvykle vektor příznaků. Vektor příznaků je definován *jako* $x = (x_1, x_2, x_3, \dots x_N)$, kde každý příznak x_M je definován číslem. Vektor příznaků je vstupem do klasifikátorů, jež jsou tvořeny diskriminačními funkcemi. V případě klasifikace do více tříd je diskriminačních funkcí více. Diskriminační funkce se značí jako $g(x)$, kde x je vektor příznaků. Výstupem diskriminační funkce je číslo, jež udává, jak moc vstupní vektor náleží do klasifikované třídy. Pokud bude klasifikátor tvořen z více diskriminačních funkcí, tak vstupní objekt bude zařazen do třídy, ke které má nejblíže (diskriminační funkce vrátí největší hodnotu). Výstupní třída se poté označuje jako $d(x)$ [8].

Specifikem těchto klasifikátorů je to, že diskriminační funkce $g(x)$ jsou neměnné a je v nich zapouzdřena veškerá logika klasifikace. Není tedy možné klasifikátor použít pro klasifikaci do jiné třídy, než pro kterou byl vytvořen. To je jistá nevýhoda těchto druhů klasifikátorů.

[4] RGB - Red Green Blue model



Obrázek 2: Klasifikátor s diskriminačními funkcemi.

Diskriminační funkce pro rozpoznávání obdélníku bude jistě hledat v obrázku objekt, jež je tvořen čtyřmi kolmými hranami. Kdežto diskriminační funkce pro rozpoznání trojúhelníku bude hledat v obraze pouze tři hrany, jež tvoří uzavřenou oblast. Princip činnosti těchto funkcí je dosti odlišný a není možné změnou vstupních konfiguračních dat diskriminační funkce pro rozpoznávání obdélníku přetransformovat diskriminační funkci pro rozpoznávání trojúhelníků. Diskriminační funkce je obvykle popsána algoritmicky a změna takového algoritmu není možná beze změny kódu.

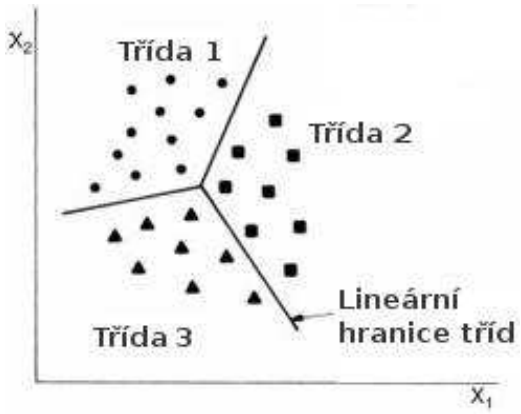
Řešením, které se v dnešní době používá stále častěji, je použití obecné diskriminační funkce, která mění své chování na základě konfiguračních dat. Klasifikátory v tomto případě obsahují pouze jednu inteligentní diskriminační funkci. Učení v takovémto případě umožňuje jednou napsaný klasifikátor použít pro klasifikaci různých objektů. Samozřejmě různé druhy obecných klasifikátorů (diskriminačních funkcí) se lépe hodí na různé druhy klasifikace. Takový klasifikátor je však nutné nejprve naučit rozpoznávat správnou množinu dat.

Učící klasifikátory mají obvykle dvě etapy svého života. V první etapě se klasifikátor učí klasifikovat, na základě trénovacích dat. V druhé etapě již klasifikátor pracuje v běžném (klasifikačním) režimu a klasifikuje vstupní objekty do výstupních tříd. Proces učení bývá obvykle složitější a výpočetně náročnější než proces klasifikace. Avšak tento problém většinou nevadí, jelikož na naučení klasifikátoru máme dostatek času. Avšak naučený klasifikátor je zpravidla rychlý, což je jedna z požadovaných vlastností pro klasifikátory.

Neučící klasifikátor je zpravidla pomalejší, jelikož musí zkoumat větší prostor vektoru příznaků a provádět tak více testů na příznacích. Učícímu algoritmu stačí, když se naučí, které příznaky jsou významné pro danou třídu objektů, a jaké mají mít rozsahy hodnot. Na základě těchto znalostí je poté schopen rychle rozhodovat a klasifikovat objekty do několika tříd.

Učení klasifikátoru, jak již bylo napsáno, není jednoduché a může se k němu přistupovat několika způsoby. Prvním způsobem je takzvané učení s učitelem. Dalším způsobem je učení bez učitele. Někdy se v literatuře uvádí také pojem učení s nedokonalým učitelem [8].

Učení bez učitele – *unsupervised learning* předpokládá na vstupu množinu vstupních dat, o kterých však nic neví. Vstupní data se snaží klasifikovat do několika tříd na základě jejich podobnosti. Zdali se naučí klasifikátor klasifikovat vstupní data správně, záleží na vstupních datech. Pokud budou vstupní data špatná (špatně separovatelná), nemusí se klasifikátor naučit, ve fázi učení, klasifikovat vstupní objekty správně. Příkladem algoritmu, jenž se učí bez učitele, je shluková analýza (K-Means [37]). Snahou klasifikátoru je klasifikovat vstupní data do několika tříd na základě podobnosti dat. Nejpodobnější objekty jsou klasifikovány do stejné množiny a prostor je tak rozdělen do několika tříd. Na obrázku 3 je zobrazen výsledek klasifikace algoritmu K-Means, pro k = 3 (počet tříd).



Obrázek 3: K-Means, výsledek klasifikace [41].

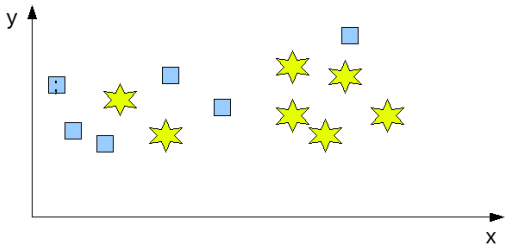
Učení bez učitele je obvykle náročné a musí probíhat v několika iteracích, ve kterých se postupně zlepšuje výsledek klasifikace. Nevýhodou tohoto přístupu je, že není vhodný pro učení obecných klasifikátorů.

Nejvíce užívanou metodou je učení s učitelem – *supervised learning*. U této metody se předpokládá, že na vstupu klasifikátoru je množina dvojic $(\vec{x}, Y)$, kde $\vec{x} = (x_1, x_2, ..x_N)$ je vektor příznaků klasifikovaného objektu (vstupní objekt) a *Y* je hodnota, jež říká, do které třídy patří daný objekt. Během učení se periodicky prochází uvedená vstupní množina dat a klasifikátor se na ní snaží naučit se klasifikovat vzorové objekty do správných tříd. Během procesu učení dochází k budování vnitřní struktury klasifikátoru, nebo k nastavování vnitřních hodnot klasifikátorů. Důležitým milníkem při učení klasifikátorů je zastavení učení ve správný čas. Pokud by učení probíhalo příliš dlouhou dobu, mohlo by dojít k přeučení klasifikátoru. To znamená, že by správně klasifikoval jen objekty ze vzorové (učící) množiny a všechny jiné objekty by vždy klasifikoval jako nevyhovující dané klasifikované třídě. Takový klasifikátor by byl poté bezcenný. Pro určení konce učení se většinou používá výpočtu chyby při klasifikaci. Chyba může být představována například jako počet špatně klasifikovaných dvojic trénovací množiny $(\vec{x}, Y)$.

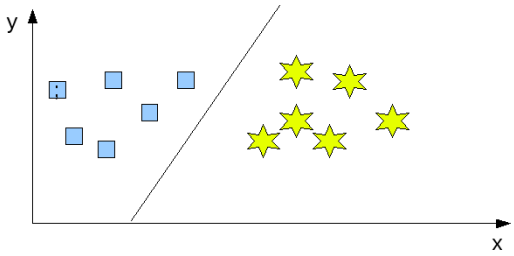
Pro správné natrénování klasifikátoru je důležité, aby trénovací množina byla dostatečně reprezentativní. To je aby obsahovala dostatečný počet objektů, ale také aby objekty zahrnovaly co největší variaci možných vstupních dat. Pokud bychom například konstruovali klasifikátor na rozpoznávání obličejů, musí trénovací množina obsahovat obličeje lidí s různými výrazy (smějící se tvář, zamračená tvář, neutrální výraz a podobně). Pokud by tomu tak nebylo, výsledný klasifikátor by poté klasifikoval správně například jen neutrální obličeje.

2.4 Lineární klasifikátor

Lineární klasifikátor pracuje pouze s daty, která jsou lineárně separovatelná. To znamená, že data lze rozdělit na dvě separátní části pomocí hyperplochy. Pokud budeme uvažovat *N* rozměrný prostor, data se nám podaří separovat pomocí *N-1* rozměrné hyperplochy. Například pro dvou-rozměrný prostor je lineární separátor přímka.



Obrázek 4: Lineárně neseparovatelný prostor



Obrázek 5:Lineárně separovatelný 2D prostor

Na obrázku 4 je prostor s objekty, jež nejsou lineárně separovatelné. Data nelze rozdělit jednou přímkou na čtverce a hvězdy. Avšak na obrázku 5 již lze objekty rozdělit do dvou různých tříd pomocí přímky, prostor je tedy lineárně separovatelný. V případě dvourozměrného prostoru by byly vstupní hodnoty tvořeny pomocí dvou souřadnic v prostoru *x* a *y*. Výstupní hodnota klasifikátoru by byla buď ano, patří do množiny, nebo ne, nepatří do množiny. Diskriminační funkce je v dvourozměrném prostoru rovnicí přímky. A určuje, zda bod leží nad přímkou nebo pod přímkou [17].

2.5 Slabé klasifikátory

Slabým klasifikátorem může být jakákoliv funkce, která má výsledky klasifikace lepší než náhodná funkce. To znamená, že výsledek klasifikace musí být lepší než 0,5 (více než 50% objektů musí klasifikovat správně). Slabý klasifikátor má tedy obecně malou úspěšnost v rozpoznávání. To však nevadí, jelikož se téměř nikdy nepoužívá samostatně. Slabé klasifikátory (*weak classifier*) bývají spojovány do složitějších systémů, které se nazývají silné klasifikátory (*strong classifier*).

Jako základ slabých klasifikátorů v obrazovém zpracování velmi často slouží obrazové příznaky. Příznaky můžeme rozdělovat na spojité a diskrétní. Spojité mohou vracet jakoukoliv reálnou hodnotu, jejich obor hodnot je neomezený. Diskrétní příznaky mohou vracet jen hodnoty z množiny přirozených čísel. Diskrétní příznaky lze převést na spojité, ale ne všechny spojité příznaky můžeme převést na diskrétní příznaky.

Spojité příznaky představují konvoluci na různých místech v obraze a o různé velikosti. Mezi spojité příznaky patří například Haarovy a Gáborovy vlnky. Každý příznak je definován svým tvarem (bází konvolučního jádra) a pozicí v obraze. Diskrétní příznaky jsou definovány podobně jako spojité. Jsou definovány pomocí pozice v obraze a velikostí mřížky (viz následující podkapitola).

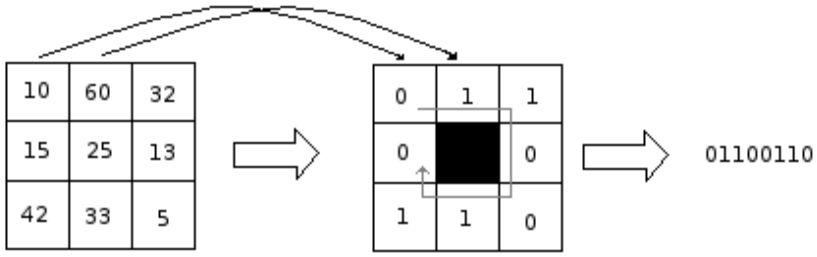
Příznakově založené klasifikátory jsou pro zpracování obrazu výhodnější, jelikož jsou univerzálnější než pixelově založené systémy. Zároveň i rychlost systémů založených na příznacích bývá obvykle vyšší než u pixelově založených systémů. Flexibilnost takového systému je též na vyšší úrovni.

V dalších podkapitolách práce budou popsány některé významné druhy spojitých a diskrétních příznaků a také budou popsány klasifikátory na nich založené. Tyto klasifikátory se často používají například pro detekci obličeje v obraze [16], [29], [33].

2.5.1 Local Binary Pattern (LBP)

Local Binary Pattern (LBP) je druh příznaků, který spadá do kategorie diskrétních příznaků. Algoritmus byl představen v pracích [25] a později v [22]. LBP je strukturní operátor pro analýzu obrazu, který převádí strukturu vstupního obrazu do výstupního obrazu, jež je složen ze stupně šedi. LBP vytváří stupně šedi pro každý pixel tak, že vezme jeho nejbližší osmi-okolí a porovná každou

hodnotu z osmi-okolí se středem (právě zpracovávaným pixelem). Výstupem z porovnání každého pixelu se středem je binární hodnota 1 nebo 0. 1 pokud je pixel z osmi-okolí větší než středový (referenční) pixel, nebo 0 pokud je pixel z osmi-okolí menší než středový pixel. Tímto způsobem dostaneme 8 bitů, které budou tvořit novou hodnotu pixelu ze středu porovnání.

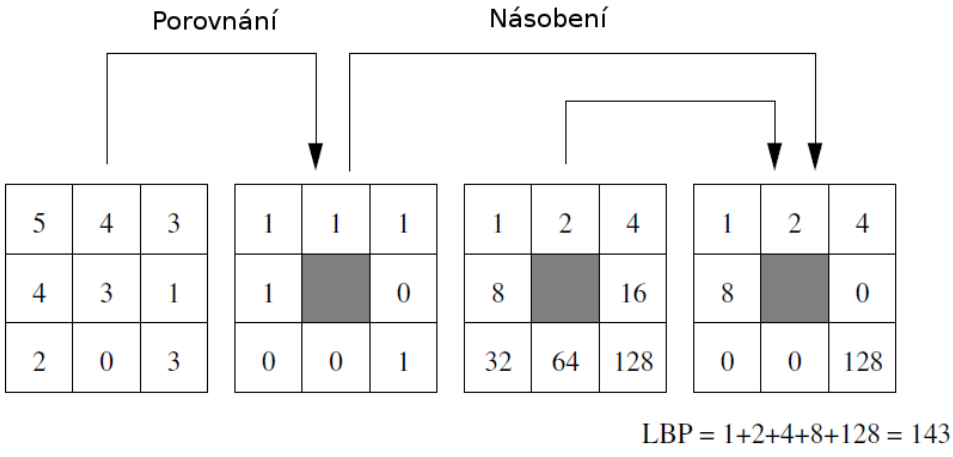


Obrázek 6: Schéma LBP klasifikace

Na obrázku 6 je naznačeno schéma základního typu LBP klasifikátoru. Středový pixel zde má hodnotu 25. Na levé části obrázku je naznačen původní rastrový obraz převedený do číselné formy (například sečtené hodnoty RGB modelu). Středová hodnota se poté porovnává s každým pixelem osmi-okolí. Výsledek porovnání je vidět ve střední části obrázku.

Po porovnání se provádí poslední část LBP klasifikátoru, to je převod příznaků (hodnot) získaných při porovnávání, na jednu výslednou hodnotu. V tomto případě má výsledná hodnota (příznak) osm bitů. Výslednou hodnotu složíme ze získaných příznaků tak, že začneme se zpracováváním v levém horním rohu a budeme pokračovat ve směru hodinových ručiček. Jednotlivé bity skládáme do výsledného osmi-bitového čísla. Na obrázku je proces naznačen šedou šipkou. Výsledný příznak, který získáme, má hodnotu 01100110 (v binární podobě). Pokud jej převedeme do dekadické soustavy, bude mít hodnotu 102. Pořadí čtení příznaků může být v literatuře různé například [25] používá jiné pořadí. Například se může postupovat po řádcích a středový pixel se vynechá, jak je uvedeno na obrázku 7. Pořadí skládání výsledné hodnoty z výsledků porovnání není významné a tak neovlivní celkový proces. Ovšem to jen za předpokladu, že bude skládání bitů v daném algoritmu vždy stejné (tj. při učení klasifikátoru i při samotné klasifikaci).

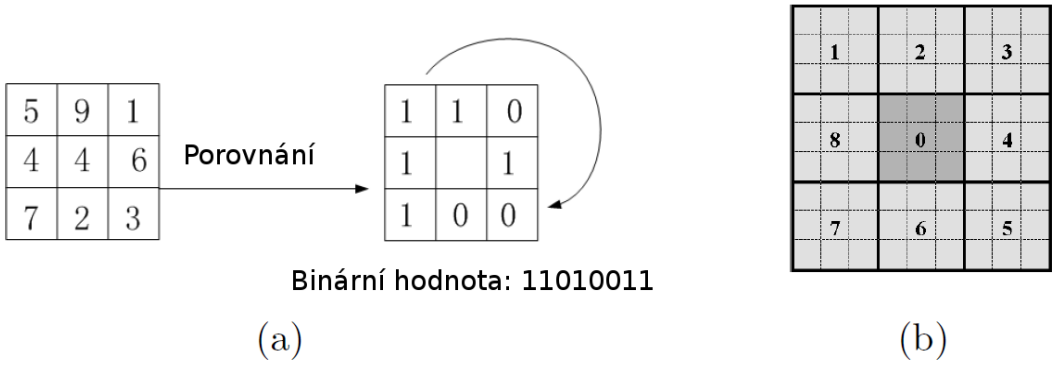
V [25] a [22] není použito pro získání výsledné hodnoty skládání bitů, ale používá se maska pro násobení.



Obrázek 7: LBP získání výsledné hodnoty násobením [22].

Jak je vidět z obrázku 7, dochází k násobení výstupu porovnání (druhý obrázek z leva) s předdefinovanou maskou (třetí obrázek z leva). V tomto případě je změněno pořadí významnosti pixelů, pixely se procházejí po řádcích. Výsledek po násobení je na obrázku vpravo. Výslednou hodnotu příznaku získáme sečtením políček matice, což je 143.

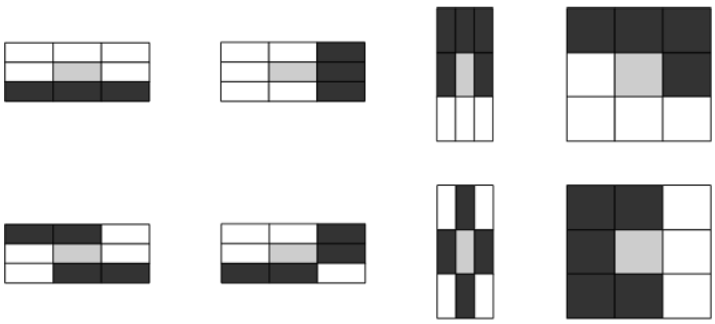
LBP je robustní algoritmus. Na výsledky klasifikace v rastrovém obraze nemá vliv, s jakou intenzitou světla je snímek pro klasifikaci pořízen. Avšak pro získání jeho větší robustnosti byly vyvinuty dokonalejší metody. Jendou z metod vylepšení původního LBP je MB-LBP - *Multi-scale Blok Local Binary Pattern* [21]. MB-LBP spočívá v nahrazení jednoho pixelu v mřížce 3x3 původního LBP za regiony, to je za několik sousedících pixelů. Například region (subregion) o rozměru dva krát dva body. Toto vylepšení činí algoritmus mnohem robustnějším. Algoritmus se nemusí zaměřovat na mikrostruktury v obraze, ale může brát v úvahu i makrostruktury, které lze nyní jednoduše vytvářet pomocí změny souřadnic subregionů a také změny velikosti jednotlivých regionů. Obraz také může být počítán velmi rychle pomocí integrálního obrazu [14], [21] (více o integrálním obraze je v kapitole 2.5.5).



Obrázek 8: MB-LBP a subregiony [21]

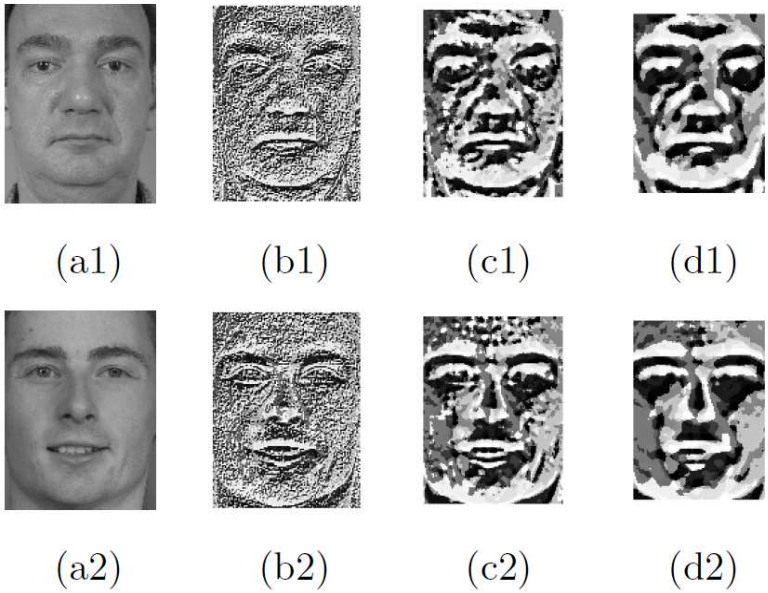
Na obrázku 8 (b) je ukázáno, jak dochází k vytváření subregionů. Základní mřížka pro LBP je tvořena *3 x 3* subregiony, kde každý subregion je tvořen oblastí v obraze *3 x 3* pixelů. Obrázek 8 (b) představuje *9 x 9* MB-LPB konfiguraci. Reprezentativní hodnota subregionu vznikne sečtením, případně konvolucí, pixelů v subregionu. Poté se celý blok MB-LBP zredukuje na původní LBP blok *3 x 3* a dále se ve zpracování postupuje stejně jako u LBP.

Pro MB-LBP lze používat různé typy mřížek, několik základních je zobrazeno na obrázku 9. Základní mřížka má vždy rozměr *3 x 3*, ale subregiony mohou mít rozdílné rozměry.



Obrázek 9: Typy mřížek pro MB-LBP [40]

Výsledný obraz po provedení MB-LBP je zobrazen na obrázku 10. Obrázek *(a)* představuje původní obrázek před provedením LBP. Obrázky *(b)*, *(c)* a *(d)* jsou výsledky po provedení MB-LBP s různými konfiguracemi subregionů.



Obrázek 10: Výsledek obrázku po provedení MB-LBP [40]
(b) 3x3 MB-LBP, (c) 9x9 MB-LBP, (d) 15x15 MB-LBP.

LBP metoda, ať již v základní nebo rozšířené verzi, se velmi často používá. Jeví se jako perspektivní metoda, která může být efektivně implementována i v FPGA technologii. Často se používá pro statickou strukturní analýzu, ale lze je použít i pro dynamickou strukturní analýzu. Časté použití je také v detekci obličeje [14], [40]. Další možnosti využití LBP může být nalezeno v [24], [26], [30].

2.5.2 Local Rank Differences (LRD)

Local Rank Differences – LRD je další příznakovou metodou používanou ke klasifikaci, především obrazových dat. Patří do množiny diskrétních příznakových metod. Hlavní výhodou LRD je její inherentnost k šedotónovým transformacím v obraze. Metoda se dokáže vypořádat s obrazem, který má vysoký jas, ale i s obrazem, který má nízkou hodnotu jasu. LRD dosahuje při klasifikaci podobných výsledků jako Haarovy příznaky (viz. kapitola 2.5.5).

Předpokládejme skalární obraz reprezentovaný následující funkci: $I(x,y) \rightarrow \mathbb{R}$ (funkce má jako parametr pozici pixelu pomocí souřadnic *x* a *y*, návratovou hodnotou je pixel obrazu). Vzorkovací funkce v takovém obraze je definována následujícím vzorcem.

$$S_{xy}^{mn}(u,v) = \frac{1}{mn}\sum_{i=1}^{m-1}\sum_{j=1}^{n-1} I(x+m(u-1)+i, y+n(v-1)+j) \qquad (2.1)$$

- $x, y, m, n, u, v, i, j \in \mathbb{Z}$
- *m,n* – dimenze vzorkovacího bloku
- *x,y* – pozice pixelu v obrázku, od kterého se provádí vzorkování

Funkce podvzorkovává každý pixel původního obrazu pomocí několika nových pixelů a to v každém směru. Tento způsob je založen na sčítání obdélníkových oblastí. Nyní byl uveden jeden

způsob definice vzorkovací funkce, ale není to jediný způsob. Další možný způsob je pomocí konvolucí oblastí obrazu s vhodným jádrem vlnkového filtru.

Na představených vzorkovacích funkcích je založena následující obdélníková maska.

$$M_{xy}^{mnwh} = \begin{bmatrix} S_{xy}^{mn}(1,1) & S_{xy}^{mn}(2,1) & \cdots & S_{xy}^{mn}(w,1) \\ S_{xy}^{mn}(1,2) & S_{xy}^{mn}(2,2) & \cdots & S_{xy}^{mn}(w,2) \\ \vdots & \vdots & \ddots & \vdots \\ S_{xy}^{mn}(1,h) & S_{xy}^{mn}(2,h) & \cdots & S_{xy}^{mn}(w,h) \end{bmatrix} \tag{2.2}$$

Parametry masky *m* a *n* určují dimenze vzorkovacího bloku. Parametry *x* a *y* určují pozici v původním obrázku tak jako v definici vzorkovací funkce *S*. Parametry *w* a *h* představují dimenzi masky. Experimenty bylo zjištěno, že vhodný rozměr masky pro detekci objektů, pomocí AdaBoost (viz sekce 2.6.2) je *3 x 3* (*w = 3, h = 3*). Tento rozměr masky je stejný jako u LBP (viz kapitola 2.5.1). LRD se tak do jisté míry podobá LBP. Pro jiné klasifikační účely je však vhodné změnit velikost masky a experimentálně zjistit nejlepší vhodný rozměr. Jako dimenze vzorkovacích funkcí se často používají masky o rozměrech 1x1, 2x2, 2x4, 4x2 pixelů.

Pro každou pozici v masce lze vypočítat *rank* pomocí následujícího vzorce.

$$R_{xy}^{mnwh}(u,v) = \sum_{i=1}^{w}\sum_{j=1}^{h}\begin{cases}1 & jeli\ S_{xy}^{mn}(i,j) < S_{xy}^{mn}(u,v) \\ 0 & jinak\end{cases} \tag{2.3}$$

Rank je tomto případě dán pořadím členů masky, ve které jsou všechny členy seřazené dle velikosti – rostoucím způsobem. LRD obrazový příznak je definován vzorcem.

$$LRD_{xy}^{mnwh}(u,v,k,l) = R_{xy}^{mnwh}(u,v) - R_{xy}^{mnwh}(k,l) \tag{2.4}$$

Celý zápis může být mírně usnadněn, pokud jej zapíšeme ve vektorovém tvaru. V tomto případě budou řádky matice nahrazeny vektory.

$$V_{xy}^{mnwh} = \begin{bmatrix} S_{xy}^{mn}(1,1) & S_{xy}^{mn}(2,1) & \cdots & S_{xy}^{mn}(w,h) \end{bmatrix} \tag{2.5}$$

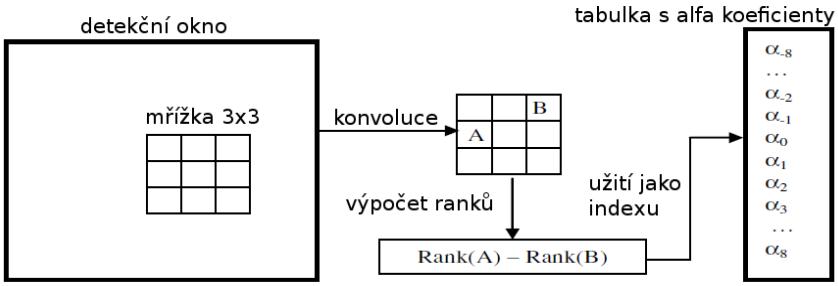
Výpočet ranku bude mít poté tvar (poznamenejme, že $V_{xy}^{mnwh}(i)$ značí i-tý člen vektoru).

$$R_{xy}^{mnwh}(a) = \sum_{i=1}^{w\cdot h}\begin{cases}1\ jeli\ V_{xy}^{mn}(i) < V_{xy}^{mn}(a) \\ 0 \qquad\qquad jinak\end{cases} \tag{2.6}$$

Poté je výpočet LRD dvou pozicí *a, b* v rámci vektorů.

$$LRD_{xy}^{mnwh}(a,b) = R_{xy}^{mnwh}(a) - R_{xy}^{mnwh}(b) \tag{2.7}$$

Tato podkapitola je částečně převzata z [27].



Obrázek 11: LRD klasifikátor [27]

Na obrázku 11 je ukázka LRD klasifikátoru. Detekční okno má velikost *31 x 31* pixelů a dimenze masky LRD klasifikátoru je *3 x 3* (M_{xy}^{mn33}). Každý subregion masky však může představovat více pixelů. Výsledná hodnota okna v rámci masky je počítána pomocí konvolucí s předdefinovanými maskami a poté je vypočítána pomocí vzorkovací funkce *S*. Taková předefinovaná maska může vypadat například následovně.

$$h_{2x2} = \begin{bmatrix} 1/4 & 1/4 \\ 1/4 & 1/4 \end{bmatrix}, h_{4x2} = \begin{bmatrix} 1/8 & 1/8 & 1/8 & 1/8 \\ 1/8 & 1/8 & 1/8 & 1/8 \end{bmatrix}, h_{wxh} = \begin{bmatrix} 1/wh & \cdots & 1/wh \\ \vdots & \ddots & \vdots \\ 1/wh & \cdots & 1/wh \end{bmatrix}$$

LRD je navržený tak, aby jej bylo možné implementovat v FPGA a ASIC technologii. Výstup LRD klasifikátoru je podobný výstupu klasifikátoru založeného na Haarových příznacích (viz kapitola Haarovy příznaky 2.5.5). Haarovy příznaky produkují rozdíl intenzit obrazu dvou oblastí a LRD produkuje rozdíl dvou ranků. Výhoda LRD spočívá v tom, že je nezávislý na celkové intenzitě obrazu. Nevadí mu, pokud je obraz tmavější nebo světlejší, kdežto u Haarových příznaků se musí provádět normalizace intenzity obrazu.

V obraze velikosti 24 x 24 pixelů (velikost okna používaného pro detekci obličeje) můžeme vygenerovat celkem 304704 příznaků LRD s mřížkou velikosti 3x3 buňky. Kdežto ve stejném obraze můžeme vygenerovat jenom 86400 příznaků založených na Haarových vlnkách. To dává LRD jistou výhodu při používání v silných klasifikátorech, kdy při fázi trénování má trénovací proces větší možnost výběru slabých klasifikátorů.

V projektu [39] autoři vytvořili s pomocí LRD klasifikátorů silný klasifikátor založený na AdaBoost. LRD klasifikátory v tomto projektu byly implementovány v FPGA. A tak bylo dosaženo velké rychlosti.

2.5.3 Local Rank Patterns (LRP)

LRP je založen na znalostech získaných při práci s LRD klasifikátorem. Jeho snaha je zaměřena na zlepšení klasifikačních schopností LRD klasifikátoru. LRP je tedy zčásti stejný jako LRD klasifikátor. Klasifikátory používají stejný princip pro výpočet ranků. Ale následná manipulace s ranky se již liší. V následujícím textu bude představen postup výpočtu LRP klasifikátoru.

Pro popis funkčnosti klasifikátoru předpokládejme skalární obraz, který můžeme popsat funkcí $f: \mathbb{Z}^2 \to \mathbb{R}$**.** Dále předpokládejme vzorkovací funkci ve tvaru ($\boldsymbol{x}, \boldsymbol{u} \in \mathbb{Z}^2, g: \mathbb{Z}^2 \to \mathbb{R}$):

$$S_{\boldsymbol{x}}^{g}(\boldsymbol{u}) = (f * g)(\boldsymbol{x} + \boldsymbol{u}) \tag{2.8}$$

Parametr g vzorkovací funkce určuje typ konvolučního jádra, které se použije pro výpočet. Vektor $\boldsymbol{x}$ představuje souřadnice pro vzorky. Pro další výpočet zavedeme vektor relativních souřadnic ($n \in \mathbb{N}$):

$$\boldsymbol{U} = [\boldsymbol{u}_1\ \boldsymbol{u}_2\ \ldots\ \boldsymbol{u}_N], \boldsymbol{u}_i \in \mathbb{Z}^2 \tag{2.9}$$

Tento vektor je tvořen dvou-dimenzionálními souřadnicemi. Určuje okolí počítaného příznaku. Využívá se společně se vzorkovací funkcí (2.8). Po dosazení vzorkovací funkce do vektoru $\boldsymbol{U}$ dostaneme nový vektor, který je tvořen výsledky vzorkovací funkce pro okolí udané pozicí $\boldsymbol{x}$:

$$\boldsymbol{M} = \left[S_{\boldsymbol{x}}^{g}(u_1)\ S_{\boldsymbol{x}}^{g}(u_2)\ \ldots\ S_{\boldsymbol{x}}^{g}(u_n)\right] \tag{2.10}$$

Vektor $\boldsymbol{M}$ si můžeme představit jako takzvanou masku, ze které se budou vypočítávat ranky. Pokud bychom se vrátili zpět k LRD, tak představuje vzorec (2.5). Výpočet ranků se provádí například pomocí vzorce:

$$R_k = \sum_{i=1}^{n} \begin{cases} 1, if\ M_k < M_i \\ 0, jinak \end{cases} \tag{2.11}$$

Výpočet výsledného příznaku, který je indexem do tabulky natrénovaných hodnot, se provede pomocí vzorce:

$$LRP(a,b) = R_a \cdot n + R_b$$

Parametr *n* představuje počet vzorků, které tvoří vektor **M**. Tím se zajistí unikátní výsledek pro každé dva různé ranky R_a a R_b. Výsledek se také může zapsat jako dvojice ve tvaru:

$$LRP(a,b) = [R_a\ R_b] \tag{2.12}$$

LRP nemusí být tvořeno jen pomocí dvou ranků, ale může být tvořeno obecně více ranky. Pak je možné výsledek reprezentovat pomocí n-tice hodnot, jako v rovnici (2.12). Nebo se výsledný rank, pokud jej chceme reprezentovat jako jedno číslo, vypočte dle následujícího vzorce (vzorec je uveden pro čtveřici):

$$LRP(a,b,c,d) = R_a \cdot n^3 + R_b \cdot n^2 + R_c \cdot n^1 + R_d \cdot n^0 \tag{2.13}$$

Výpočet je částečně přebrán z [15].

LRP má některé vlastnosti stejné jako LRD. Například zůstala nezměněna vlastnost invariantnosti vzhledem k intenzitě jasu obrazu. LRP, má oproti LRD, mnohem větší diskriminativní sílu popisu vlastností obrazu. Je to způsobeno tím, že na svém výstupu může reprezentovat mnohem více stavů. LRP bylo testováno v [15], kde se ukázaly být lepšími než LRD příznaky a tradiční příznaky založené na Haarových vlnkách. Lepší výsledky byly dosahovány jak v úspěšnosti klasifikace, tak i v počtu příznaků (slabých klasifikátorů) potřebných k úspěšné klasifikaci. Jak LRD, tak i LRP se dají efektivně implementovat v FPGA technologii.

2.5.4 Local Rank (LR)

LR příznak je základem pro výpočet LRD i LRP příznaků. Výstup LR příznaku je stejný jako výpočet jednoho ranku v případě výpočtu LRP příznaku. Tento příznak je, jako oba navazující, invariantní vzhledem k intenzitě jasu obrazu. Počet stavů, kterých může výstup LR klasifikátoru nabývat je malý. Pro masku o devíti položkách může nabývat pouze devíti hodnot. Diskriminativní popis oblasti obrázku tak není příliš silný. Výpočet příznaku se provede například pomocí rovnic (2.8), (2.9), (2.10), (2.11). Výstup rovnice (2.11) je výstupem LR příznaku.

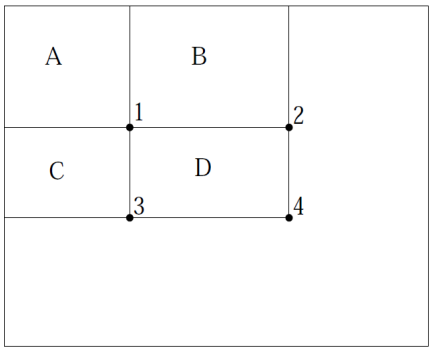
2.5.5 Haarovy příznaky

Haarovy příznaky jsou založeny na Haarových vlnkách, jež byly definovány Alfredem Haarem v roce 1909 v práci [9]. Na Haarových vlnkách jsou založeny mnohé klasifikační systémy. Haarovy vlnky většinou tvoří základ slabých klasifikátorů. A z těchto slabých klasifikátorů se následně sestavují silné klasifikátory. Jako první použili Haarovy příznaky ve své práci Viola and Jones, kteří pracovali na systému detekce tváří [32]. Haarovy příznaky jsou velmi univerzální a používají se ve velké míře při detekci objektů v obraze. S Haarovy příznaky lze pracovat velmi efektivně, pokud je vstupní obraz v takzvaném integrálním tvaru [32].

Integrální obraz je nový název pro tzv. sumační obraz. Nový název použili poprvé ve své práci Viola and Jones. Každý pixel integrálního obrazu musí být pro nově vzniklý integrální obraz vypočítán dle definovaných vztahů. Výpočet však není složitý, pokud jej provádíme inteligentně (s využitím již vypočtených pixelů). Každý pixel integrálního obrazu je definován vzorcem.

$$ii(x, y) = \sum_{x' \leq x, y' \leq y} i(x', y') \tag{2.14}$$

Kde *ii(x,y)* představuje integrální obraz a *i(x',y')* představuje původní obraz. Již ze vzorce je vidět, že prvek integrálního obrazu je tvořen součtem všech pixelů nalevo a nahoru od právě počítaného pixelu. Pokud spočítáme takový integrální obraz, můžeme poté vypočítat sumu pixelů určité oblasti velmi efektivně. Při výpočtu bude potřeba maximálně jen čtyř přístupů do paměti. Toto tvrzení plyne z obrázku 12, na kterém je naznačen výpočet sumy oblasti integrálního obrazu. Obrázek představuje jeden snímek obrazu a jsou v něm naznačeny čtyři oblasti *A*, *B*, *C* a *D* a čtyři významné body *1*, *2*, *3* a *4*.

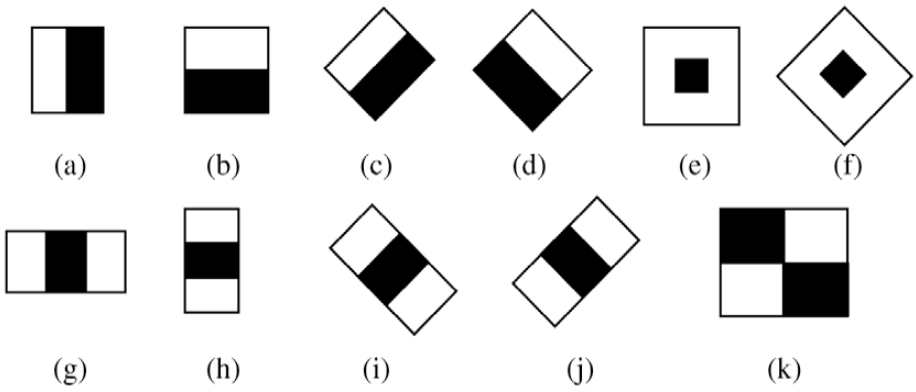


Obrázek 12: Integrální obraz [32]

Suma všech pixelů v oblasti *A* je právě hodnota integrálního obrazu v bodě *1*. Suma všech pixelů v oblasti *B* se vypočte jako rozdíl pixelů *2* a *1*, tedy *2-1*. Pro oblast *C* platí podobný výpočet a to *3-1*. Pro oblast *D* je výpočet nejsložitější a požaduje se tedy nejvíce přístupů do paměti. Výpočet se tedy provede jako *4 – 3 – 2 + 1*.

Byl popsán výpočet integrálního obrazu a nyní se vrátíme zpět k popisu Haarových vlnek. Nevýhodou používání příznaků založených na Haarových vlnkách je to, že je třeba mít normalizovanou intenzitu obrazu, aby se mohly příznaky efektivně používat. K normalizaci se často používá derivace intenzity obrazu. Tato operace není problematická na běžných počítačích, ale tvoří problém při zpracování na FPGA. Avšak v práci [4] byl vytvořen systém na detekci obličejů pomocí Haarových příznaků v FPGA.

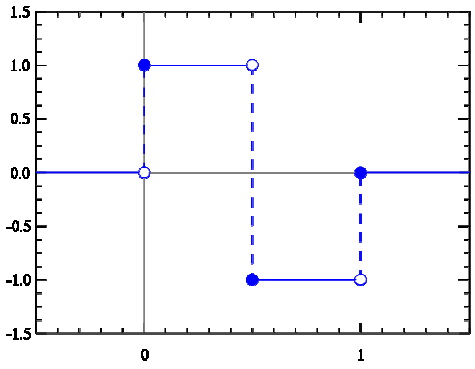
Jak již bylo napsáno v úvodu podkapitoly, Haarovy vlnky byli definovány v [9]. Na obrázku 13 je naznačeno několik základních typů Haarových vlnek [20], [35].



Obrázek 13: Haarovy vlnky [20]

Výpočet Haarových příznaků z vlnek je založen na odečítání sum pixelů z bílé oblasti a z černé oblasti. Na obrázku 13 jsou zobrazeny i rotované Haarovy vlnky (c, d, f, i, j) tyto se však většinou nepoužívají v kombinaci s integrálním obrazem, jelikož by výpočet sum oblastí pomocí integrálního obrazu nešel použít.

Jen pro úplnost doplním, že Haar vlnky původně definoval jen v jednom rozměru (1D), a až následně byli převedeny do dvourozměrného prostoru. Původní vlnka v 1D vypadá následovně (obrázek 14).

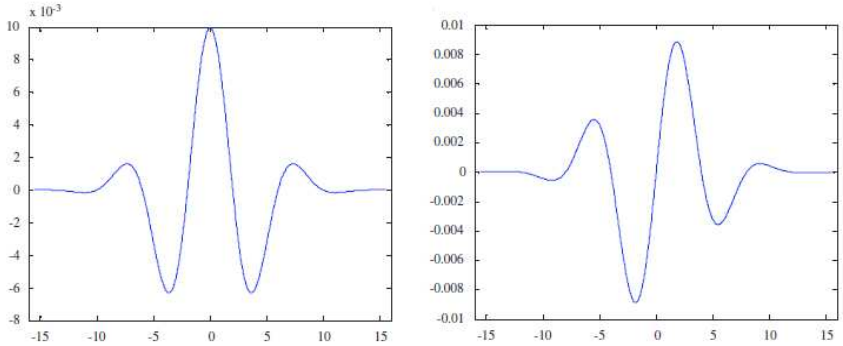


Obrázek 14: Haarova vlkna v 1D prostoru

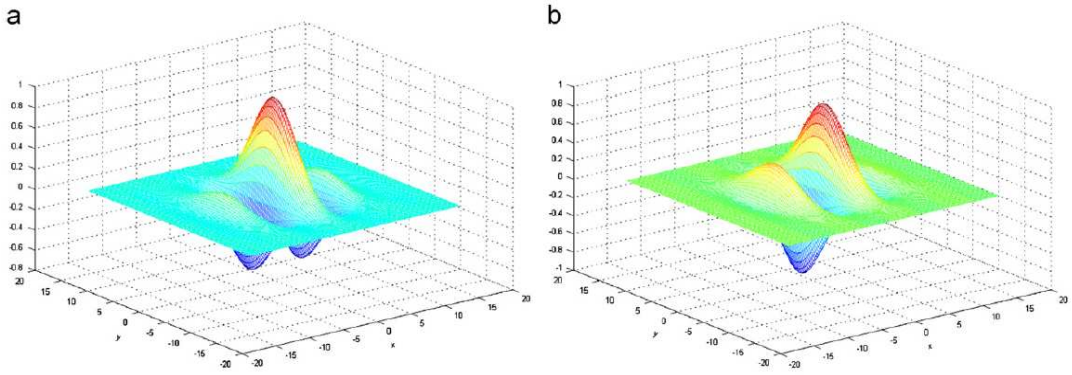
2.5.6 Gaborovy příznaky

Gaborovy příznaky jsou založeny na Gaborových vlnkách, které byly popsány v [19]. Jejich významnou vlastností je velká schopnost popisu vlastností obrazu. Avšak výpočet založen na Gaborových vlnkách je velmi náročný na čas, a proto se využívají jen v časově nenáročných aplikacích. Gáborovy příznaky poskytují rozumný kompromis mezi frekvenčním a prostorovým rozlišením obrazu. Další výhodou je, že Gaborovy příznaky jsou podobné vnímání obrazu člověka pomocí šedé kůry mozkové. A tak se nám jeví i zpracování obrazu pomocí Gaborových příznaků přirozené. Na obrázku 15 jsou 1D Gaborovy vlnky. Na obrázku 16 jsou stejné vlnky jako na obrázku 15, ale jsou zobrazeny v 2D prostoru. Gaborovy vlnky na obrázcích mají exponenciální změny

průběhů. Stejně jako funkčnost Haarových příznaků, tak i Gaborovy příznaky jsou závislé na normalizaci intenzity osvětlení obrazu.



Obrázek 15: Gaborovy vlnky v 1D [5]



Obrázek 16: Gaborovy vlnky v 2D [5]

2.6 Silné klasifikátory

V anglické literatuře se nazývají *strong classifiers*. Jsou tvořeny spojením několika slabých klasifikátorů. Množina slabých klasifikátorů může být tvořena různými klasifikátory například LDR, LBP, klasifikátory založené na Haarových příznacích nebo na Gaborových příznacích. Případně může být slabý klasifikátor tvořen jednoduchou neuronovou sítí. Všechny slabé klasifikátory dostávají obvykle na vstup stejná data a na základě těchto dat produkují svůj výstup. Výstup různých slabých klasifikátorů je poté váhován (normalizován), dle jejich významnosti při klasifikaci. Výslednou hodnotu silného klasifikátoru může tvořit například vážená suma výstupních hodnot slabých klasifikátorů. A na základě této hodnoty se silný klasifikátor rozhodne, zda obrázek patří do klasifikované množiny nebo ne. Silné klasifikátory jsou obvykle založeny na učení. Během učení se tedy snaží naleznout množinu slabých klasifikátorů, které budou mít po ukončení učení dobré výsledky při klasifikaci vstupních objektů do výstupních tříd.

2.6.1 Boosting

Pojem boosting je spojen se skupinou učících klasifikačních algoritmů. Základní myšlenkou boostingu je kombinace jednoduchých a poměrně špatných klasifikátorů (slabých klasifikátorů) do

jednoho silného klasifikátoru, který však vykazuje velmi dobré výsledky při klasifikaci. Mnoho metod boostingu je založeno na lineární kombinaci slabých klasifikátorů. V rámci silného klasifikátoru se každému slabému klasifikátoru, v něm obsaženém, přiřazuje jeho významnost při rozhodovacím procesu [6]. Kořeny boostingu sahají k projektu PAC (*Probably Approximately Correct*), jenž se zabýval strojovým učením. První metody měly mnoho stinných stránek a tak vzniklo několik jejich vylepšení, kterých bylo uvedeno hned několik. Například AdaBoost, WaldBoost a další.

2.6.2 AdaBoost

Název AdaBoost vychází z názvu *adaptive boosting*. Je to tedy vylepšení boostingu. Vylepšení spočívá v přizpůsobení silného klasifikátoru na vstupní data. Přizpůsobení se provádí pomocí procesu učení. Při kterém si AdaBoost vybere množinu slabých klasifikátorů a nastaví jim případně požadované parametry pro správnou klasifikaci.

AdaBoost byl představen autory Freund a Schapire [7] v roce 1995. Tento algoritmus řeší mnoho nevýhod původního boostingu, ze kterého vychází.

Učení AdaBoostu probíhá opakovaně v $t = 1,2,\ldots T$ krocích. T je dáno počtem slabých klasifikátorů, které může AdaBoost využívat. V každé iteraci jsou slabé klasifikátory používány s jinými vahami D_t souboru vstupních dat *S*. Úlohou každé iterace je naleznout nejlepší slabý klasifikátor z množiny dostupných klasifikátorů *H*. Hledá se taková hypotéza $h_t : X \rightarrow \{-1,+1\}$, která minimalizuje chybu klasifikátoru s aktuálními vahami D_t:

$$\varepsilon_t = P_{i \sim D_t}[h_t(x_t) \neq y_t] \tag{2.15}$$

Nejlepší slabý klasifikátor h_t je poté zařazen do silného klasifikátoru s koeficientem α_t, který je vybrán s ohledem na vypočtenou chybu ε_t a vahami D_t.

$$\alpha_t = \frac{1}{2}\ln\left(\frac{1-\varepsilon_t}{\varepsilon_t}\right) \tag{2.16}$$

Výsledný silný klasifikátor *H* je lineární kombinací vybraných slabých klasifikátorů.

$$H(x) = sign\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right) \tag{2.17}$$

Po každém vybrání slabého klasifikátoru, do množiny silného klasifikátoru, se musí přepočítat soubor vah D_t. D_{t+1} je vypočítáno z původního D_t a nově vybraného klasifikátoru h_t. Hodnoty vah se mohou jak zvětšovat, tak i zmenšovat.

$$D_{t+1}(x_i) = \frac{D_t(x_i) \cdot \exp(-\alpha_t y_i h_t(x_i))}{Z_t} \tag{2.18}$$

kde $Z_t = \sum_{i=1}^{m} D_t(x_i) \cdot \exp(-\alpha_t y_i h_t(x_i))$.

Změna vah D_t je jednou ze základních myšlenek AdaBoostu, jelikož se pomocí ní vyjadřuje, jak dobře dopadla klasifikace v předešlých iteracích procesu učení. Váha $D_t(x_i)$ vstupního vzorku *i*, v kroku učení *t* vyjadřuje, jak dobře proběhla klasifikace v předešlých krocích učení pomocí všech, do té doby vybraných, slabých klasifikátorů. Níže je uveden kompletní algoritmus [7].

Mějme $S = \langle (x_1, y_1), \ldots, (x_m, y_m) \rangle$, kde $x_i \in X$ vstupní data a $y_i \in Y = \{-1, +1\}$

Inicializace vah $D_1 = \frac{1}{m}$

For t=1,…,T:

1. Nalezni $h_t = \arg\min_{h_j \in H} \varepsilon_j$; $\varepsilon_j = \sum_{i=1}^{m} D_t(x_i) I[h_j(x_i) \neq y_i]$

2. If $\varepsilon_t \geq 1/2$ then stop

3. $\alpha_t = \frac{1}{2} \ln\left(\frac{1-\varepsilon_t}{\varepsilon_t}\right)$

4. Aktualizuj váhy

$$D_{t+1}(x_i) = \frac{D_t(x_i) \cdot \exp(-\alpha_t y_i h_t(x_i))}{Z_t}$$

kde $Z_t = \sum_{i=1}^{m} D_t(x_i) \cdot \exp(-\alpha_t y_i h_t(x_i))$.

Výsledný klasifikátor má poté tvar:

$$H(x) = sign\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$$

Algoritmus AdaBoost **[7].**

Vstupem algoritmu je trénovací množina dat a označení každého jejího prvku, zda má být klasifikován pozitivně nebo negativně. Na počátku algoritmu se inicializují váhy na počáteční hodnotu, která je nepřímo úměrná počtu vstupních dat. Poté se provádí cyklus, který se skládá ze čtyř kroků. V prvním kroku se hledá takový slabý klasifikátor, který má nejmenší váženou chybu na trénovacích datech. (Funkce *I* vrací *1*, pokud je podmínka splněna, nebo *0* pokud podmínka není splněna.) V druhém kroku se kontroluje základní podmínka pro slabé klasifikátory v AdaBoostu a to je, že chyba slabého klasifikátoru musí být menší než 0,5. Podmínka je důležitá vzhledem ke konvergenci algoritmu.

Ve třetím kroku se vybírá koeficient α_t pro lineární kombinaci slabých klasifikátorů v rámci silného klasifikátoru. Výpočet α_t je udělán tak, aby se minimalizoval horní odhad chyby výsledného klasifikátoru.

$$\varepsilon_{tr}(H) \leq \prod_{t=1}^{T} Z_t = \frac{1}{2^T} \prod_{t=1}^{T} \sqrt{\varepsilon_t (1-\varepsilon_t)} \tag{2.19}$$

Aktualizace vah v kroku čtyři slouží ke zmenšení vah dobře klasifikovaných vstupních dat a ke zvětšení vah špatně klasifikovaných dat. Tím je zajištěno, že v dalším kroku se bude hledat klasifikátor, který lépe klasifikuje doposud špatně klasifikované vstupní data.

Výhodou AdaBoostu je, že velmi rychle konverguje k řešení silného klasifikátoru, který má malou chybu na trénovacích datech. Samozřejmě tento předpoklad je platný za podmínky, pokud se podaří naleznout nové slabé klasifikátory, které mají chybu menší než 0,5 (tj. úspěšnost je větší než náhodná funkce).

2.6.3 WaldBoost

Algoritmus WaldBoost byl poprvé představen autory Šochman a Matas v [29]. WaldBoost kombinuje AdaBoost s Wald's sekvenčním pravděpodobnostním testem. Algoritmus se pokouší řešit problém velkého počtu vyhodnocení slabých klasifikátorů při použití AdaBoostu. Pokud je klasifikátor tvořen AdaBoostem, je předem známo, kolik vyhodnocení slabých klasifikátorů se musí provést, abychom získaly výsledek silného klasifikátoru. Při vyhodnocování pomocí metody WaldBoost je sice znám maximální počet slabých klasifikátorů, ale není známo, kolik se jich pro aktuální vstupní řetězec vyhodnotí.

Pro každé detekční okno, nad kterým algoritmus pracuje, se vyhodnotí jiný počet slabých klasifikátorů. Pro takový způsob výpočtu však musíme po každém vyhodnocení slabého klasifikátoru porovnávat aktuální výslednou hodnotu silného klasifikátoru (součet všech doposud získaných výsledků slabých klasifikátorů) se dvěma prahy. První práh určuje hodnotu, nad kterou je výsledek klasifikace považován za pozitivní. Druhý práh určuje hodnotu, pod kterou je výsledek klasifikace hodnocen jako negativní. Po každém vyhodnocení slabého klasifikátoru se musí tedy porovnávat doposud získaný výsledek silného klasifikátoru se zadanými prahy. Celkový počet dvojic prahů pro vyhodnocení musí být stejný počet, jako je počet slabých klasifikátorů. Jednotlivé prahy se nastavují při procesu trénování klasifikátoru.

Rychlost klasifikátoru založeného na WaldBoost metodě je vyšší než klasifikátoru založeného na AdaBoost metodě. (AdaBoost musí vyhodnotit vždy všechny slabé klasifikátory, ze kterých se skládá.) Přesnost WaldBoost klasifikátoru a AdaBoost klasifikátoru je téměř shodná. Za předpokladu, že budou správně nastaveny parametry prahů u WaldBoost metody. Jejich špatné nastavení by vedlo k chybné funkci celého klasifikátoru.

3 FPGA

Před příchodem technologie FPGA byla jediná možnost tvorby hardwarově akcelerovaných algoritmů pomocí speciálních, mnohdy jednoúčelových, jednotek. Tyto jednotky mohou být tvořeny například pomocí ASIC technologie (zde se myslí první generace ASIC). Vývoj takové ASIC jednotky je však příliš nákladný a trvá dlouhou dobu. Tím se prodlužuje čas uvedení výrobku na trh (*time-to-market*). Proto se může stát, že aplikace nebude vyvinuta dostatečně rychle a na trhu již o ni nemusí být zájem.

Cena vývoje ASIC technologie je velmi vysoká, jelikož obsahuje mnoho fází vývoje. Od textové formulace problému, přes popis algoritmu v hardware, až po vytvoření masky pro výrobu ASIC čipu. Nevýhodou také je, že pokud na již vyrobeném čipu chceme provést nějaké vylepšení, nebo ještě hůře opravu, tak v takovém případě většinou není možnost čip modifikovat a musí se podstoupit celý proces výroby ASIC obvodu od počátku a čip poté hardwarově vyměnit. Vývoj jednoho ASIC čipu stojí od několika stovek tisíc dolarů, u jednodušších čipů, až do několika milionů dolarů, u složitých čipů. Avšak výhodou ASIC technologie je, že pokud je již čip kompletně navržen a vyrábí se ve velkých sériích, je výroba takového čipu poměrně nenákladná (jen několik dolarů). U ASIC technologie se také často setkáváme se systémy SoC (*Systém on Chip*). Takový čip obsahuje nejenom samotné výpočetní jádro ale i některé vybrané podpůrné obvody, jako mohou být paměťové moduly, některé obvody pro ovládání periferií a podobné podpůrné obvody.

Jak již bylo napsáno, technologie ASIC má sice několik výhod, ale také má řadu nevýhod. Proto přišla na svět jiná technologie zvaná FPGA (*Field-programmable gate array*). Tato technologie například umožňuje znovupoužití čipu pro jiný druh výpočtu. K znovupoužití stačí pouze nahraní nového *designu* (funkčnosti) do FPGA čipu. Tato věc byla u ASIC technologie nemožná. Vývoj aplikací pro FPGA je jednodušší, a tak není třeba tolik času pro vývoj aplikace jako u ASIC technologie. Z toho plyne, že cena vývoje aplikací pro FPGA bude nižší. Nevýhodou FPGA čipu je jeho vysoká pořizovací cena. Ta mnohonásobně převyšuje cenu ASIC čipu. Avšak pokud vyrábíme aplikaci, která poběží jen na několika přístrojích, tak se tato technologie jeví velmi perspektivní. Navíc bude takový systém možné udržovat a nadále jej vylepšovat bez nutnosti změny hardware.

Myšlenka programovatelných obvodů se poprvé objevila již na počátku osmdesátých let dvacátého století. Tehdy byla vyvinuta technologie PAL (*Programmable Array Logic*). PAL obvody byly založeny na programovatelném poli AND[5] prvků, následovaným fixním polem OR[6] prvků. Umožňovaly realizovat jen několik logických funkcí. Tyto obvody však bylo možné konfigurovat jen jednou. V roce 1984 se prosadily v obvodech EPROM[7] paměti, a tím byla umožněna jejich opakovaná rekonfigurace. Mazání takové paměti však probíhalo pomocí UV (ultrafialového) záření, což bylo mnohdy zdlouhavé. Pro jejich označení se vžila zkratka PLD (*Programmable Logic Device*).

Další rozvoj technologie vedl k CPLD (*Complex Programmable Logic Device*). V této technologii byl snížen příkon a byla zvýšena hustota tranzistorů na čipu. CPLD obsahuje několik jednotek PLD, které spojuje pomocí programovatelného propojení. První takový obvod byl uveden na trh firmou Altera v roce 1985. U této technologie byly použity elektronicky mazatelné paměti EECMOS[8].

[5] AND – prvek pro logické násobení

[6] OR – prvek pro logický součet

[7] EPROM – Errasable Programmable Read-Only Memory – Přepisovatelné paměti typu ROM

[8] EECMOS – Electrically Errasable Complementary Metal Oxide Semiconductor – Elektricky mazatelné paměti založené na technologii CMOS.

Ve stejném roce přišla firma Xilinx s prvním obvodem FPGA. Kolem roku 1993 se začaly do FPGA přidávat paměti RAM a tím se rozšiřovala jejich funkčnost. Následujícím krokem ve vývoji bylo zvětšování počtu logických hradel. V dnešní době se na čipu FPGA vyskytuje až několik milionů hradel.

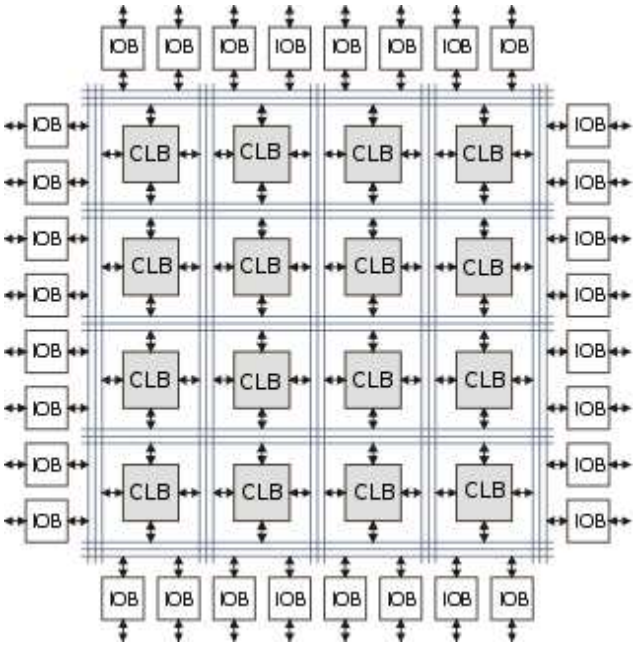
Obvody FPGA ve srovnání s obvody ASIC jsou obvykle větší a méně výkonné. Také mají zpravidla větší příkon. Avšak všechny tyto nevýhody jsou vyváženy jejich rekonfigurovatelností.

Vytváření obvodů (návrhů) pro technologii FPGA probíhá v dnešní době především pomocí specializovaných nástrojů (syntetyzátory). K popisu funkčnosti takového obvodu se používá převážně jazyků pro popis hardware jako je například rodina jazyků HDL (*Hardware Description Language*), jmenovitě VHDL (*VHSIC hardware description language; VHSIC: very-high-speed integrated circuit*), případně jazyka VERILOG nebo Handel-C. Jazyk VHDL se používá převážně v Evropě a jazyk VERILOG v Americe. Jazyk Handel-C poskytuje z uvedených jazyků největší abstrakci nad hardwarovými prvky. Avšak čím vyšší je abstrakce v programovacím jazyku, tím musí být program pro převod do cílové platformy (syntetizátor) složitější.

3.1 Princip funkce FPGA

FPGA (*Field-programmable gate array*) jak již bylo řečeno je konfigurovatelný obvod. Své konfigurovatelnosti dosahuje pomocí konfigurovatelných bloků, z nichž je složen. A také pomocí konfigurovatelného propojení základních stavebních bloků. Každý ze základních konfigurovatelných bloků může být naprogramován na jednu ze základních funkcí jako je AND a XOR, nebo z něj může být sestaven obvod pro vykonávání matematických funkcí, nebo dekódování. V FPGA jsou obvykle obsaženy ještě další prvky, jako jsou například paměti nebo násobičky či jednoduché mikroprocesory. Tyto obvody může návrhář využívat dle své potřeby.

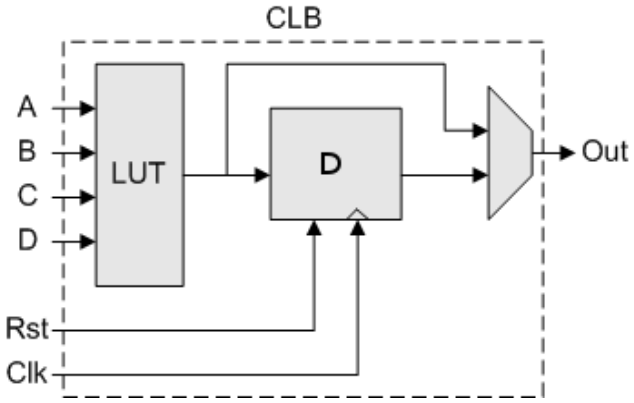
FPGA se v dnešní době skládá až z několika desítek milionů (viz rodina Virtex-6) ekvivalentních hradel. Obvykle se jedná o dvou-vstupá hradla NAND.



Obrázek 17: Struktura FPGA čipu.

Legenda: IOB – Vstupně-výstupní obvody, CLB – Konfigurovatelné bloky.

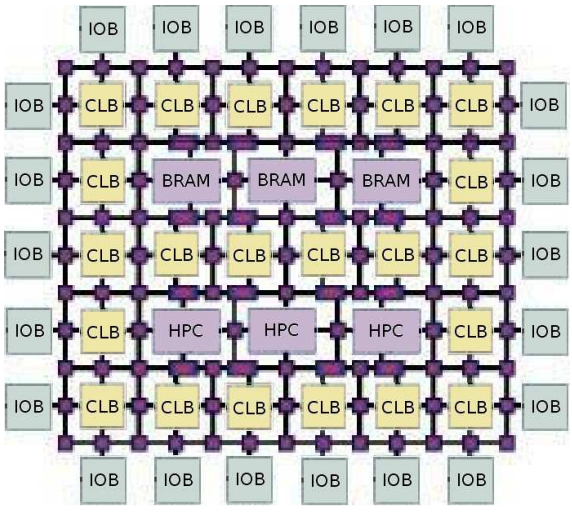
Na obrázku 17 je zobrazena základní struktura FPGA čipu. V architektuře jsou použity tři základní stavební bloky. Prvním a nejvýznamnějším z nich je CLB (*Configurable Logic Block*). Jedná se o konfigurovatelné bloky, které tvoří velkou část funkcionality výsledné aplikace. Velký podíl na činnosti aplikace také nese programovatelná logická síť, pomocí níž lze propojit jak bloky CLB tak, i IOB bloky. Propojení CLB bloků může být téměř libovolné. Obvody IOB (Input/Output Block) jsou obvody pro zajištění vstupu a výstupu dat z čipu FPGA. Pro každý pin FPGA čipu je jeden IOB obvod. Tento obvod obsahuje obvykle paměťový registr, multiplexor, budič a ochranné obvody. Lze říci, že obsahuje obvody pro napěťové přizpůsobení obvodů mimo čip FPGA s obvody uvnitř FPGA. Základní struktura obvodu CLB je zobrazena na obrázku 18.



Obrázek 18: Struktura CLB bloku

Legenda: LUT – Vyhledávací tabulka, D – paměťový prvek.

CLB se skládá z LUT (*Look Up Table*), což je vyhledávací tabulka, paměťového prvku D a výstupního multiplexoru. CLB tak tedy může sloužit jen jako paměťový registr. CLB může mít až čtyři vstupní hodnoty, což nám dává celkem $2^4 = 16$ kombinací. Vyhledávací tabulka má tedy celkem *16* hodnot a může se s ní implementovat logická funkce, která má až *4* vstupní hodnoty. Pokud potřebujeme realizovat funkci s více než čtyřmi vstupními hodnotami, musíme použít více bloků CLB. V dnešní době se používají složitější bloky CLB. Na obrázku 18 je zobrazen jenom základní typ CLB. Stejně tak i struktura moderních čipů FPGA je rozdílná od struktury uvedené na obrázku 17.



Obrázek 19: Struktura novějších čipů FPGA

Na obrázku 19 je zobrazena pokročilejší struktura čipu FPGA. Z obrázku lze vidět, že do čipu FPGA přibyly nové prvky. Jedním prvkem je BRAM (*Block Random Access Memory*), což je rychlá synchronní statická asociativní paměť. Druhým novým prvkem na obrázku je HPC (*Hardware performance computing*). Tato zkratka značí prvky, jež umožňují zrychlit výpočet. Například hardwarová násobička, nebo dokonce celý mikroprocesor. Zkratka HPC byla zavedena v této práci a tak se neshoduje se zkratkami v literatuře. V literatuře se může někdy shodovat se zkratkou DSP (*Digital Signal Procesing*).

Dalšími prvky, jež nejsou na obrázku, ale jsou přítomny na čipu FPGA jsou například obvody pro obnovení charakteristik hodinového signálu – PLL (*Phase Lock Loop*). Případně obvody pro dělení či násobení hodinového signálu DLL (*Delay Lock Loop*).

3.2 Současné FPGA čipy

V současné době přichází na trh nová rodina FPGA čipů společnosti Xilinx pod názvem Virtex-6 [1]. Je to již šestý pokračovatel úspěšné řady Virtex. Rodina začíná jednoduchým čipem XC6VLX75T a pokračuje až k největšímu čipu XC6VLX760. Různé druhy čipů této rodiny mají různé periferie a jsou tak určeny pro různé použití. Rodinu čipů Virtex-6 lze charakterizovat následujícími prvky:

- Rozdělena do tří podkategorií, které se navzájem liší výbavou čipů.
- Šesti-vstupé LUT.
- Obsahují od 74496 do 758784 logických buněk v závislosti na čipu.
- 36 kB Block RAM, které lze konfigurovat i jako FIFO (*First In First Out*), šířka přístupu k datům až 72 bitů
- Nové vylepšené hardwarové násobičky a sčítačky (DSP48E1)
- Integrované bloky pro PCI-Expres sběrnici (až 5 Gb/s)
- Integrované síťové rozhraní (až 2.5Gb/s)

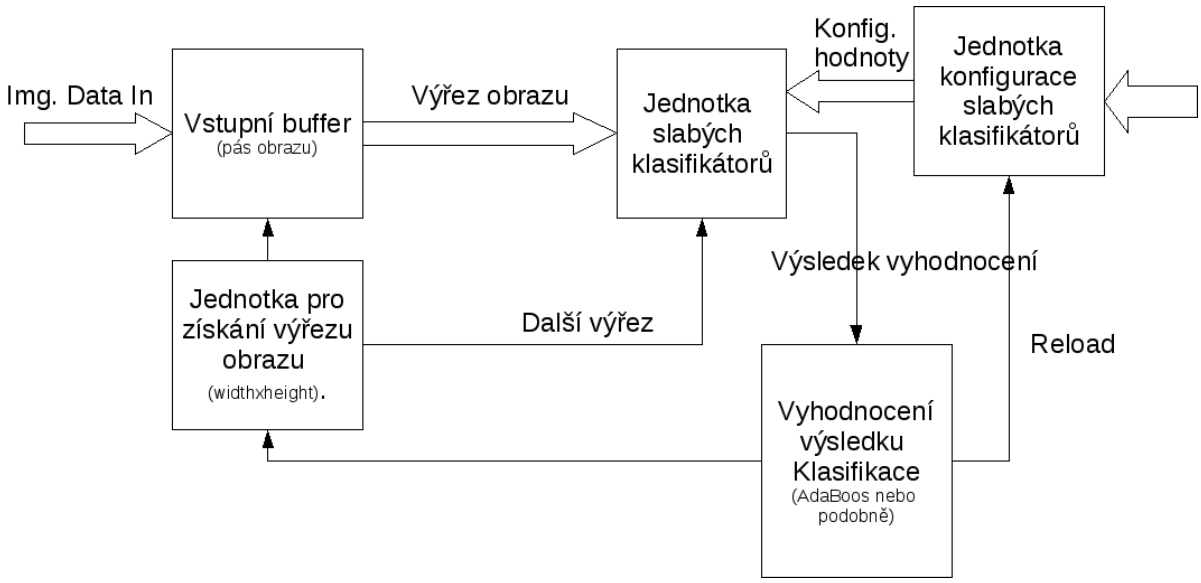
Informace jsou převzaty z [1].

4 Architektury klasifikátorů

V této kapitole budou popsány architektury klasifikátorů, které byly v rámci projektu navrženy. Ne všechny architektury vedly k úspěšnému řešení a dokončení jejich návrhu v FPGA. Jako slabé klasifikátory se předpokládají LBP, LRP nebo LR jednotky (viz kapitola 2).

4.1 Paralelní struktura klasifikátorů

Paralelní struktura klasifikátorů byla navržena s ohledem na možnost paralelně zpracovávat data pomocí několika samostatných jednotek v jednom čase. Tato architektura je navržena ve dvou úrovních popisu. První úroveň popisu se zaměřuje na zpracování vstupního obrazu a jeho ukládání do paměti v FPGA. Architektura je naznačena na obrázku 20.



Obrázek 20: Schéma paralelní architektury 1. úrovně

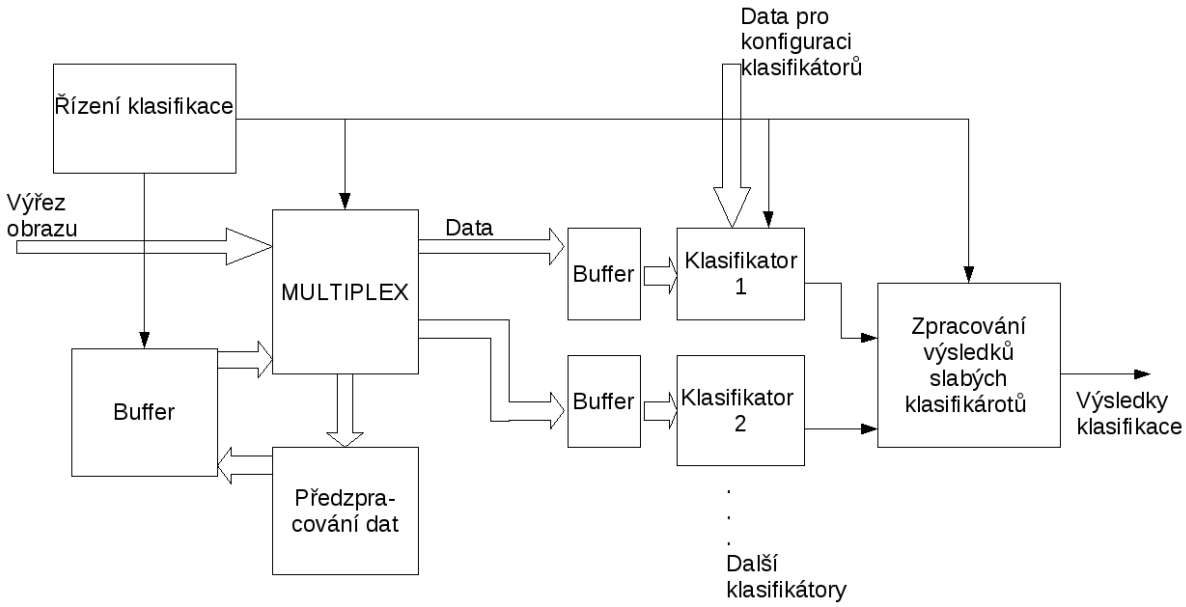
Data do architektury vstupují v bodě označeném jako *Img. Data In*. Vstupní data jsou ihned ukládána do paměti. Pro konstrukci paměti se předpokládá využití více portové paměti *BlockRAM*, aby bylo možné na jedné straně neustále ukládat data a na straně druhé je ihned číst. Jelikož není paměť pro ukládání obrazu příliš velká, může se ukládat v paměti jen část obrazu. Zde byl zvolen pro ukládání pás obrazu. Toto rozhodnutí plyne z požadavků klasifikátorů, které pracují na výřezu obrazu. Proto je nutné ukládat v paměti pás obrazu, který má výšku minimálně o velikosti detekčního okna, se kterým pracují klasifikátory.

Pokud i tak není na čipu dostatek paměti, musí se řešit ukládání pomocí snížení šířky obrazu. Jedním možným řešením je rozpůlit obraz na dvě poloviny (s překryvem velikosti šířky detekčního okna) a obě poloviny obrazu zpracovávat zvláště. V této kapitole však předpokládáme, že máme dostatečně velkou paměť pro uložení požadovaného pásu obrazu. Pokud by tomu tak nebylo, problém by bylo nutné řešit v jednotce, která posílá obrazová data do FPGA.

Pro řízení bufferu, který ukládá do paměti pás obrazu je v návrhu zařazena jednotka *Jednotka pro získání výřezu obrazu*. Ta se stará o to, aby byly v bufferech správná data, a zajišťuje správné ukládání dat do bufferu (kruhový buffer). Ze vstupního bufferu jsou data posílána do jednotky slabých

klasifikátorů. Tato jednotka je rozebrána v rámci druhého (podrobnějšího) stupně návrhu. Výsledek vyhodnocení klasifikátorů je posílán do speciální jednotky (*Vyhodnocení výsledku klasifikace*), která řídí konfiguraci slabých klasifikátorů a také dává pokyny jednotce, pro získání výřezu obrazu, že může již některá data vypustit z bufferu (data již nebudou znovu zpracovávána). Poslední jednotkou je jednotka pro konfiguraci klasifikátorů, ta se stará o to, aby se použil správný klasifikátor. Použijeme-li například pro klasifikaci algoritmus AdaBoost, tak pro každou klasifikaci v rámci stejného výřezu obrazu je použit klasifikátor s jinou konfigurací.

Na obrázku 21 je vyobrazena jednotka slabých klasifikátorů.



Obrázek 21: Schéma druhé úrovně paralelní architektury.

Na obrázku 21 je znázorněna škálovatelná architektura se slabými klasifikátory. Vstupem do jednotky je výřez obrazu. Výřezem obrazu se rozumí část obrazu, na které budou klasifikátory pracovat. V systému je opět jednotka pro řízení klasifikace, která určuje, co bude která jednotka provádět a také provádí řízení souběhu jednotek. V návrhu je zařazena jednotka pro předzpracování dat, kterou může představovat například hranový detektor. Tato jednotka má za úkol upravit vstupní obraz a tento upravený obraz poté poslat opět na zpracování slabými klasifikátory. Vstupní data tak mohou jít přímo (tj. bez předzpracování) do klasifikátorů, anebo mohou být předzpracována a teprve poté poslána ke klasifikaci. Samozřejmostí je, že předzpracování dat probíhá naprosto odděleně od klasifikace a tedy oba procesy mohou běžet paralelně.

Výsledný efekt tohoto procesu je, že v prvním kroku se zpracují původní obrazová data a ve druhém kroku klasifikace se zpracují upravená data. Jelikož architektura nebyla nadále rozvíjena, nepodařilo se ověřit, zda by byl tento způsob klasifikace s případným předzpracováním přínosný, nebo ne.

Architektura byla navržena s možností uspořádání několika slabých klasifikátorů tak aby pracovaly paralelně. Tato myšlenka byla založena na tom, že k rozhodnutí, zda spadá obrázek do klasifikované třídy nebo ne, s devadesáti procentní pravděpodobností správné klasifikace, postačuje jen několik výsledků slabých klasifikátorů. Pokud bychom prováděli více klasifikací najednou, mohli bychom snížit čas výpočtu.

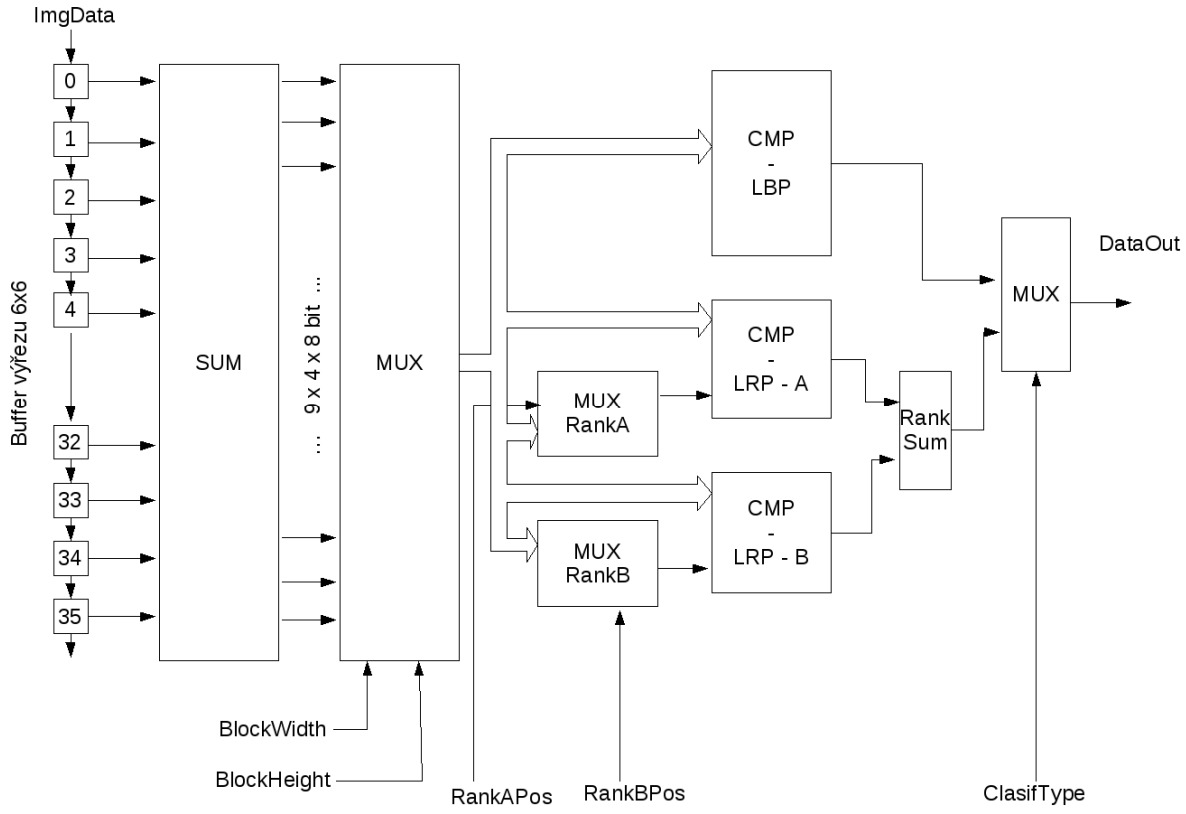
Ovšem otázkou zůstává, zda má smysl tuto technologii rozvíjet. Po několika konzultacích byla tato architektura zavrhnuta s opodstatněním, že její přínos by nebyl příliš velký s ohledem na velikost

zabraného místa na čipu FPGA. V této architektuře je totiž nutné výřez obrazu ukládat na několika místech, aby mohly klasifikátory pracovat současně. Tím by bylo na čipu zabráno příliš mnoho zdrojů jen pro vytvoření bufferů.

4.2 Klasifikátor s využitím LBP a LRP

Po zamítnutí architektury uvedené v kapitole 4.1 byla navržena nová architektura, která již neuvažuje paralelní zpracování obrazových dat pomocí několika klasifikátorů souběžně. Ale klasifikace se provádí jen pomocí jednoho klasifikátoru, který se vybere z množiny dostupných klasifikátorů v závislosti na konfiguračních datech.

Tato architektura však také nebyla nikdy realizována. Pouze bylo rozpracováno její klasifikační jádro na úrovni návrhu. Pokud bychom měli zasadit navržené klasifikační jádro do kontextu předchozí architektury, z kapitoly 4.1, představovalo by *jednotku slabých klasifikátorů* z obrázku 20.



Obrázek 22: Jádro klasifikátoru s LBP a LRP.

Na obrázku 22 je uvedeno schéma klasifikační jednotky. Tato jednotka umí provádět klasifikaci pomocí LRP a LBP příznaků. Navržená architektura pracuje nad výřezem obrazu o velikosti *6 x 6* pixelů. Vstup obrazových dat je naznačen pomocí textu *ImgData*. Obraz se uchovává ve FIFO[9] frontě o délce 36 položek (velikost výřezu, na kterém se provádí klasifikace). Jeden pixel obrazu je představován pomocí osmi-bitové hodnoty (obraz je reprezentován ve stupních šedi). Z každého stupně FIFO fronty je výstup přiveden na vstup bloku sčítaček – *SUM*. Tímto způsobem je možné získat v každém hodinovém taktu 36 hodnot pixelů. Uchovávání vstupního obrazu jen pomocí

[9] FIFO – First In First Out

zde uvedené FIFO linky není v reálném systému příliš vhodné. Zde je uvedeno jen pro jeho jednoduchost.

V bloku *SUM* se provádí sčítání hodnot pixelů jednotlivých regionů. Regiony zde představují jednotlivé konvoluce. Výsledek bloku *SUM* je vždy na osmi bitech. Pokud se v bloku *SUM* provádí sečtení dvou, nebo více pixelů, musí se výsledek normalizovat do původního rozsahu 0 až 255 (to je do rozsahu osmibitového čísla). Výsledek se musí proto vydělit počtem pixelů, které se sčítají. Velikosti regionů (konvolucí) v mřížce *6 x 6* pixelů jsou *1 x 1, 1 x 2, 2 x 1 a 2 x 2* pixelů. Součty všech regionů se provádějí paralelně a výsledné součty pro všechny regiony jsou tak známy ve stejném čase. Tím dostaneme na výstupu bloku *SUM* čtyři devítice osmibitových hodnot. Každá devítice představuje součty pro jednu matici o velikosti *3 x 3* pixelů. Taková matice je poté vstupem do slabých klasifikátorů.

Dle parametrů *BlockWidth* a *BlockHeight* se v bloku *MUX* vybírá jedna ze čtyř devític hodnot. Výstupem bloku *MUX* je již jen jedna devítice hodnot, která je přivedena na vstup komparátorů, které představují výpočet LBP a LRP klasifikátorů. U LRP klasifikátoru jsou předřazeny dva bloky *MUX RankA* a *MUX RankB*. Ty slouží k výběru hodnot, se kterými se budou porovnávat všechny ostatní hodnoty dle principu LRP a bude tak vypočítán požadovaný *rank*.

Výstup bloku *CMP LRP-A* tvoří horní čtyři bity výsledné LRP hodnoty. A výstup bloku *CMP LRP-B* tvoří spodní čtyři bity výsledku LRP klasifikace. K jejich sloučení dochází v bloku *RankSum*. Zda na výstupu klasifikátoru bude LRP nebo LBP se určuje pomocí hodnoty *ClasifType*, která je vstupem do dalšího multiplexoru *MUX*. Výstupní hodnota je v diagramu označena jako *DataOut*. Tato hodnota slouží jako index do vyhledávacích tabulek *LUT*. Vyhledávání v *LUT* nebylo v návrhu implementováno.

Představená architektura posloužila jako základ pro vybudování následující pseudo-paralelní architektury. Nevýhodou představené architektury je, že může provést jen jednu klasifikaci v čase. Avšak současně jsou v ní obsaženy dva kompletní klasifikátory (LBP i LRP), ale vždy se využije jen výsledek jednoho z nich a druhý se zahodí. Tím se zbytečně plýtvá zdroji na FPGA.

Nutnou podmínkou pro úspěšnou funkci představené architektury je existence řídící jednotky, která vyhodnocuje výsledky klasifikace a řídí klasifikační jednotku. Pokud bychom pomocí této architektury realizovali detektor objektů s detekčním oknem o velikosti *24 x 24* pixelů, museli bych mít uložen pás obrazu o výšce 24 pixelů. Což klade vysoké požadavky na paměť. Kdybychom předpokládali šířku obrazu 512 pixelů a zpracování obrazu ve stupních šedi (8 bitů na pixel), museli bychom ukládat 96 kB dat. Za předpokladu, že máme na čipu FPGA k dispozici blokové paměti RAM o velikosti 18 kB, museli bychom jich jen na uložení obrazu použít minimálně 6. Při šířce obrazu 1024 by to již bylo jedenáct paměťových modulů. Prozatím opomíjenou částí je adresovací logika, která by dokázala správně adresovat výřez potřebný pro klasifikaci. Zmíněná adresovací logika není jednoduchá a na čipu FPGA by zabrala nemalé množství zdrojů. Díky zde uvedeným důvodům byla i tato, na začátku slibně vypadající, architektura opuštěna. A byla navržena nová architektura, která do jisté míry kombinuje obě doposud uvedené architektury.

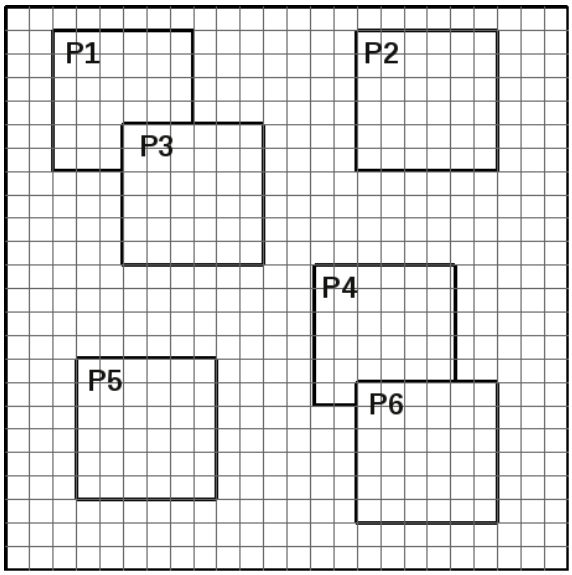
4.3 Pseudo-paralelní architektura

Jak již bylo napsáno, cílová architektura je navržena na základě znalostí získaných z předešlých návrhů. Kombinuje vybrané principy uvedených architektur se zcela novým přístupem. Pro pochopení principu navržené architektury, je nutné se nejprve seznámit s postupem zpracování obrazu v této architektuře.

4.3.1 Detekce pomocí klasifikačního okna

Již v kapitole 2 bylo napsáno, že klasifikace (rozpoznávání) probíhá v rámci detekčního okna. Například pro detekci obličejů se používá detekční okno o velikosti *24 x 24* pixelů [14]. Znamená to, že detekovaný objekt se určí jen na základě těchto pixelů. Okolní pixely jsou pro klasifikaci nepodstatné a ve výpočtu se neuvažují.

Klasifikace probíhá pomocí vyhodnocování příznaků (slabých klasifikátorů), které mohou být umístěny kdekoliv v detekčním okně. Velikosti a tvary příznaků mohou být obecně různé. Jelikož však pro realizaci v FPGA není vhodné mít velikost příznaku neomezenou, byla omezena maximální velikost na hodnotu *6 x 6* pixelů. Tento rozměr byl vybrán jako kompromis a je zcela postačující pro klasifikaci obrazu.



Obrázek 23: Detekční okno 24 x 24 pixelů s 6 příznaky.

Na obrázku 23 je znázorněno detekční okno o velkosti *24 x 24* pixelů. V detekčním okně je umístěno celkem šest příznaků. Umístění příznaků v rámci detekčního okna je výsledkem trénovacího procesu (viz kapitola 2.6). Z obrázku lze pozorovat, že se jednotlivé příznaky mohou libovolně překrývat. Každý příznak je jednoznačně určen jeho pozicí a typem. Výsledek klasifikace nad daným oknem může být dán součtem výsledků všech příznaků dle principu AdaBoost (2.6.2). Nebo se může dle principu WaldBoost (2.6.3) počítat průběžné skóre po vyhodnocení každého příznaku (slabého klasifikátoru). Pořadí vyhodnocení příznaků pro WaldBoost, ale i pro AdaBoost, je dáno procesem trénování, stejně jako typ a pozice příznaků.

Pro implementaci v FPGA, jak se později ukáže, je lepší vyhodnocení klasifikace dle principu AdaBoost. Principem urychlení, které nabízí technologie FPGA, je vypočítávat hodnoty všech příznaků v detekčním okně paralelně. Výsledky je poté možné sečíst dle principu AdaBoost a rozhodnout o výsledku klasifikace. Pro dosažení dobrých výsledků při klasifikaci je třeba vyhodnocení velkého počtu příznaků. Velikost FPGA je však omezená a většinou se nedá na jeden FPGA čip umístit potřebný počet vyhodnocovačů příznaků (slabých klasifikátorů). Proto se na čip umístí jen část slabých klasifikátorů (více v následujících kapitolách).

4.3.2 Rozdělení na hardwarovou a softwarovou část

Běžný univerzální procesor není dostatečně rychlý, aby dokázal provádět klasifikaci obrazu v reálném čase a zároveň, jak již bylo zmíněno v předchozím odstavci, není efektivní na jeden čip FPGA umístit potřebný počet slabých klasifikátorů, tak aby byl výsledek klasifikace vždy správný. Řešením tohoto problému je rozdělení klasifikační úlohy na dvě části. První z částí bude mít za úkol určit všechny detekční okna, které obsahují nebo mohou obsahovat klasifikovaný objekt, a zároveň označit co nejméně detekčních oken, která hledaný předmět neobsahují.

Druhá část klasifikace bude mít za úkol dokončit klasifikaci a označit všechna detekční okna, která obsahují hledaný objekt. Tato část může mít mnohonásobně více slabých klasifikátorů (příznaků) než předešlá, jelikož je implementována v software. Softwarová část je mnohem pomalejší než hardwarová část, ale jelikož provádí hardwarová (první) část předfiltraci detekčních oken, dochází v softwarové části ke zpracování jen zlomku detekčních oken ($1/10^3 - 1/10^5$). Tudíž pro zpracování v software postačuje běžný univerzální procesor, případně univerzální mikroprocesor.

Práce se v dalším textu zaměřuje výhradně na hardwarovou část zpracování vstupního obrazu. Implementace softwarové části není obtížná a již byla řešena v mnoha pracích [14], [36], [39].

Výhodou rozdělení klasifikátoru na dvě části je, že výsledný design nebude zabírat mnoho místa na čipu FPGA a může se tak umístit na levnější čip. Pokud bychom se na celou problematiku podívali z ekonomického hlediska, tak je velmi důležitá optimalizace návrhu vzhledem k požadovaným zdrojům. Na trhu sice existují FPGA čipy, které by dokázaly pojmout návrh kompletního klasifikátoru. Ale cena takového čipu není malá a pohybuje se v řádu tisícíců dolarů [2]. Poté by i cena celého klasifikačního zařízení byla příliš vysoká. V procesu návrhu tak bylo snahou umístit klasifikátor na co nejmenší FPGA čip. Jako vhodný se například jeví čipy z rodiny Virtex-II nebo Spartan3.

Takový čip je poměrně levný a má přijatelnou spotřebu energií. Aby měla realizace klasifikátoru v FPGA opodstatnění, předpokládá se zpracování velkého množství dat. Zdrojem dat může být například kamera s vysokým rozlišením obrazu. Řešení, které se zde nabízí je propojení kamery a čipu FPGA do jedné funkční jednotky. Taková funkční jednotka může v reálném čase snímat scénu a současně na ní provádět klasifikaci. Výstupem takové jednotky jsou dva proudy (streamy) dat. První z nich představuje původní obrazová data. Druhý by byl tvořen klasifikačními výsledky, které by označovali v původním obraze detekční okna, která v sobě obsahují hledané předměty.

Výstup uvedeného systému může být zpracováván běžným počítačem a mohou být prováděny další operace nad obrazem, které nemusí nutně souviset s klasifikací.

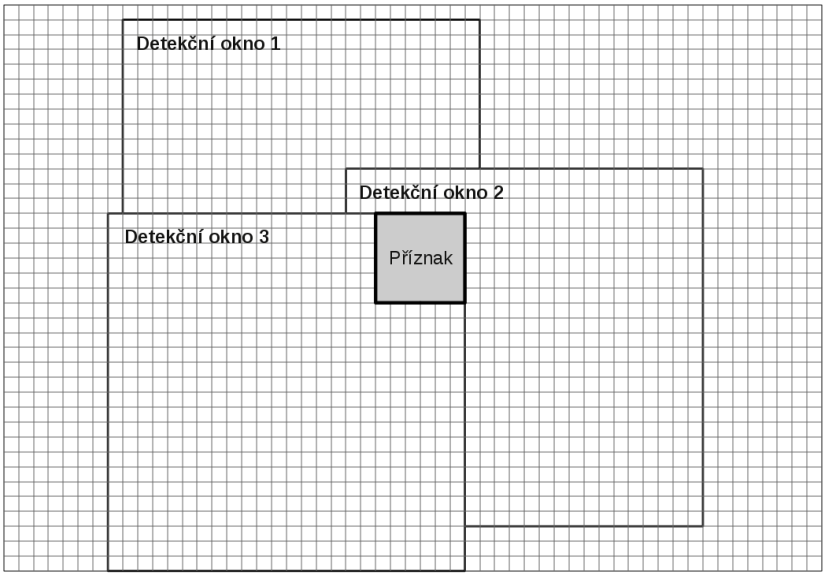
4.3.3 Princip vyhodnocení klasifikace

V kapitole 4.3.1 byla představena klasifikace pomocí detekčního okna. Pro získání výsledku klasifikace pro jedno detekční okno je potřeba vyhodnotit několik slabých klasifikátorů. Již v úvodu kapitoly bylo napsáno, že vyhodnocení probíhá paralelně. Jelikož však není vhodné uchovávat v paměti velkou část obrazu pro detekční okno o velikosti *24 x 24* pixelů (celkově se jedná o pruh obrazu o výšce 24 pixelů), přistupuje se k uchování jen malé části obrazu o velikosti *6 x 6* pixelů. Tato velikost představuje oblast obrazu potřebnou pro vyhodnocení jednoho příznaku (slabého klasifikátoru).

Na uvedeném výřezu obrazu o velikosti *6 x 6* pixelů se však provede vyhodnocení všech požadovaných klasifikátorů. Výsledky jednotlivých slabých klasifikátorů z tohoto kroku však nelze

dle principu AdaBoost sečíst a následně vyhodnotit. Pokud bychom to udělali, znamenalo by to, že všechny klasifikátory mají stejnou pozici. Tato situace však téměř nikdy nenastane.

Je proto nutné výsledky zpracovat jiným způsobem. Myšlenka zpracování výsledků spočívá ve zpoždění každého výsledku klasifikace o přesně specifikovaný čas. A jejich zpožděné sčítání.

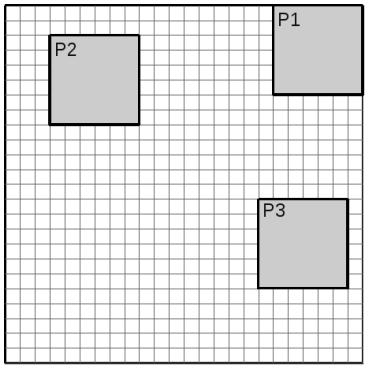


Obrázek 24: Zpracování příznaků v rámci detekčních oken.

Na obrázku 24 je znázorněno umístění aktuálně zpracovávaného příznaku v rámci tří detekčních oken. Na obrázku jsou zobrazena jen vybraná detekční okna, na kterých je provedena ilustrace přístupu zpracování. V reálném zpracování představuje každý pixel obrazu počátek nového detekčního okna. Detekčních oken je tak v obrázku velký počet. Předpokládejme, že klasifikátor na obrázku 24 je složen ze tří slabých klasifikátorů (příznaků). Příznaky jsou umístěny na pozicích *x* a *y* (souřadnice jsou udány vzhledem k detekčnímu oknu):

- P1 - *x = 19, y = 0*
- P2 - *x = 4, y = 3*
- P3 - *x = 18, y = 14*

Každé detekční okno má svou pozici pevně dánu. Během zpracování obrazu se oblast pro vyhodnocení slabých klasifikátorů (v obrázku označena jako *Příznak*) postupně posouvá pixel po pixelu přes celý obrázek. A dochází tak postupně k vyhodnocení všech slabých klasifikátorů na všech pozicích v obraze.



Obrázek 25: Detekční okno s příznaky.

Na obrázku 25 je zobrazena pozice slabých klasifikátorů (*P1, P2, P3*) v rámci jednoho detekčního okna. Výsledky všech příznaků se počítají paralelně, ale vždy na stejném místě. Proto je třeba pro získání výsledku celého klasifikátoru si postupně získané výsledky slabých klasifikátorů uložit do paměti a na konci detekčního okna, nebo po získání výsledku posledního slabého klasifikátoru, je sečíst. Tím získáme výsledek celé klasifikace.

Pokud by se měly uchovávat všechny výsledky slabých klasifikátorů až do doby, než se zpracuje poslední slabý klasifikátor, bylo by zapotřebí velké množství paměti na průběžné ukládání výsledků slabých klasifikátorů. Ze znalosti algoritmu AdaBoost je však známo, že výsledky jednotlivých slabých klasifikátorů se mohou sčítat, jakmile jsou dostupné.

Po získání výsledku klasifikátoru *P2* (z obrázku) můžeme provést sečtení jeho výsledku s výsledkem slabého klasifikátoru *P1*. Stejně tak i po získání příznaku *P3* jej můžeme sečíst s výsledkem součtu *P1+P2*. Dostaneme tak výsledek celého klasifikátoru. Výsledek dostaneme již po zpracování slabého klasifikátoru *P3* a nemusíme čekat, až okno pro výpočet příznaků projde celé detekční okno. Okamžiky, ve kterých se mají provést postupné součty slabých klasifikátorů, jsou dané jejich pozicí v rámci detekčního okna a šířkou zpracovávaného obrazu. Pro předešlý příklad a za předpokladu šířky obrazu 512 pixelů by zpoždění byly:

- Mezi *P1* a *P2* = 1521
- Mezi *P2* a *P3* = 5646

Zpoždění se získá jednoduchým výpočtem:

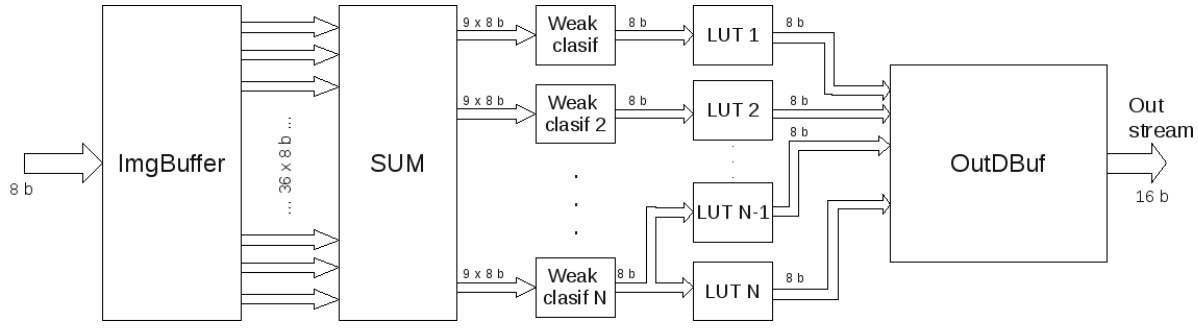
$$delay(P1,P2) = (P2.y - P1.y) * IMGWidth + (P2.x - P1.y) \quad (4.1)$$

- Kde *P1, P2* představují slabé klasifikátory,
- zápis *P1.x* představuje pozici v horizontální ose obrazu a *P1.y* ve vertikální ose obrazu.
- *IMGWidth* představuje šířku obrazu.

Pro realizaci uvedeného příkladu jsou potřeba dvě zpožďovací linky. Počet zpožďovacích linek je vždy o jednu menší, než je počet slabých klasifikátorů.

4.3.4 Popis architektury

V předchozích kapitolách byla představena základní koncepce pseudo-paralelní architektury. Nyní bude architektura představena jako celek a následně budou pospány části, ze kterých se skládá.



Obrázek 26: Schéma pseudo-paralelní architektury.

Architektura se skládá z pěti hlavních částí. První částí je blok *ImgBuffer*. Tento blok slouží pro ukládání obrazových dat potřebných pro klasifikaci. Vstupem do tohoto bloku je obraz, který vstupuje sekvenčně po pixelech. (V jednom taktu vstoupí do zpracování jeden pixel.) Pro zpracování se předpokládá obraz v odstínech šedi s 256 možnými odstíny. Výstupem bloku je třicet šest

osmibitových hodnot. Tyto hodnoty představují oblast obrazu, na které se provádí klasifikace (*6 x 6* pixelů).

Navazujícím blokem je *SUM*. Blok slouží pro výpočet konvolucí, které jsou vstupem do slabých klasifikátorů. Počet výstupů tohoto bloku není konstantní a může se pro různé konfigurace klasifikátoru měnit. Výstupy bloku *SUM* jsou vždy sdruženy po devíti osmibitových hodnotách. To je dáno principem slabých klasifikátorů, které pracují s devíti hodnotami konvolucí (viz 2.5).

Výsledky konvolucí jsou přiváděny na vstup vyhodnocovačů příznaků. Na obrázku jsou označeny jako *Weak clasif N*. Slabých klasifikátorů může být v architektuře umístěno několik. Každý pracuje samostatně a své výsledky posílá do vyhledávací tabulky – *LUT (look-up table)*. Výstup slabého klasifikátoru je maximálně osmibitový. Klasifikátory LBP, LRP, LRD produkují osmi bitový výstup. Klasifikátor LR produkuje jen čtyřbitový výstup.

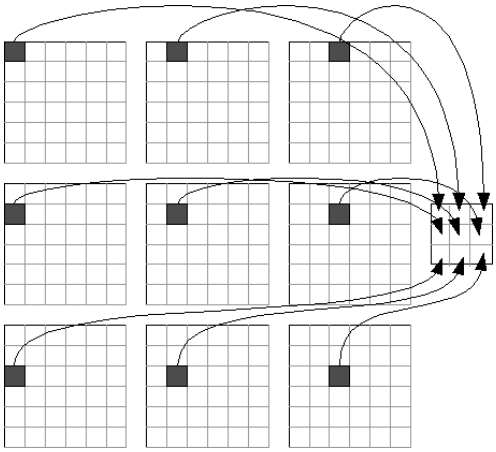
Výstup každého slabého klasifikátoru je přiveden na vstup vyhledávacích tabulek. Z obrázku je patrné, že výstup jednoho slabého klasifikátoru může být přiveden na vstup více vyhledávacích tabulek. Je to jedna z více optimalizací, vzhledem k velikosti zabraného místa na čipu FPGA. Pokud jsou zapotřebí dva slabé klasifikátory, které mají sice různou pozici a jiné koeficienty vyhledávacích tabulek, ale jsou typově shodné (oba jsou LRP nebo oba jsou LBP), lze takový slabý klasifikátor vložit do návrhu jen jednou a jeho výstup použít jako vstup do více vyhledávacích tabulek. Pokud by bylo v architektuře obsaženo 20 klasifikátorů LBP, které by pracovali se stejnou konvolucí v obraze, do výsledného návrhu by se mohla vložit jen jedna jednotka pro vyhodnocování příznaků a její výstup by se přivedl na vstup dvaceti vyhledávacích tabulek. V návrhu by se ušetřilo 19 jednotek pro vyhodnocování příznaků.

Výstup bloků *LUT* je přiveden na vstup bloku *OutDBuf*. Tento blok představuje zpracování výsledků slabých klasifikátorů. Dle principu uvedeného v kapitole 4.3.3 se musí jednotlivé výsledky ukládat do zpožďovacích linek a součet vstupních položek se musí provádět s přesně definovaným zpožděním. Zpoždění je určeno pozicí slabých klasifikátorů v detekčním okně. Výstupem této jednotky je šestnácti bitový proud dat, který pro každé detekční okno představuje výsledek klasifikace.

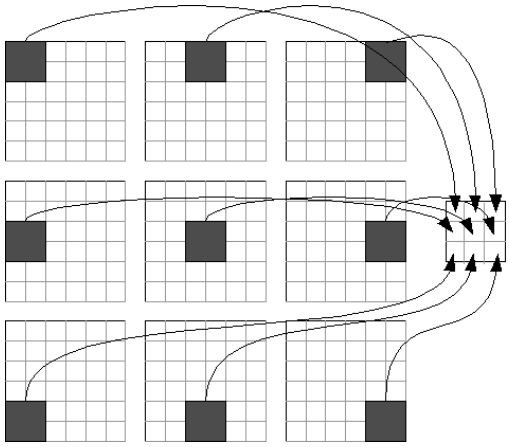
Za tuto jednotku může být připojen komparátor, který bude porovnávat výsledek součtů slabých klasifikátorů s nastaveným prahem. Práh by představoval hranici mezi pozitivní a negativní klasifikací.

4.3.5 Získání vstupu pro slabé klasifikátory

V kapitole 2.5 jsou popsány typy slabých klasifikátorů, jako jsou LBP, LRP a další. Všechny představené klasifikátory pracují na vstupu s devíticí osmibitových hodnot. Tyto hodnoty jsou obvykle reprezentovány ve tvaru matice o velikosti *3 x 3*. Matice může být naplněna hodnotami jednotlivých pixelů obrazu nebo také výsledkem konvoluce obrazu se zadanou maticí. V případě že jsou hodnoty matice tvořeny hodnotami jednotlivých pixelů, mluvíme o takzvaných vysokofrekvenčních obrazových filtrech. Pokud je matice složena z výsledků konvolucí s velkou konvoluční maticí, mluvíme o takzvaných nízkofrekvenčních obrazových filtrech.



Obrázek 27: Vysokofrekvenční filtr.



Obrázek 28: Nízkofrekvenční filtr.

Na obrázku 27 a 28 jsou zobrazeny dva typy konvolučních matic. Na každém obrázku je devět konvolučních matic (masek) a šipky, jež ukazují umístění výsledku konvoluce v rámci výsledné matice o rozměrech *3 x 3*, která reprezentuje vstup do slabých klasifikátorů. V každé konvoluční matici představuje šedé pole oblast pixelů, která se sčítá. Po provedení součtu hodnot pod šedým polem, je nutné výslednou hodnotu normalizovat. Normalizace se provádí jako celočíselné dělení součtu hodnot pixelů jejich počtem (počet pixelů pod šedou oblastí). Pro případ na obrázku 27 dělíme výsledek jedničkou. V případě obrázku 28 dělíme výsledek čtyřmi.

V případě obrázku 27 je do výsledné matice vložena vždy hodnota pixelu, která je definovaná příslušnou konvoluční maticí. V příkladu uvedeném na obrázku 28 se do políčka umístí výsledek celočíselného dělení sumy pixelů pod šedou oblastí a počtu pixelů pod šedou oblastí.

5 Implementace

V předešlé kapitole byla popsána architektura pro klasifikaci. Taktéž byla popsána základní činnost jednotlivých prvků, ze kterých se architektura skládá. Byly popsány očekávané vstupy a výstupy jednotek, obsažených v klasifikátoru. Avšak nebylo nic napsáno o způsobu vytvoření základních bloků klasifikátoru a jejich vnitřní činnosti.

AdaBoost je metoda, která vytvoří a natrénuje jeden silný klasifikátor. Ten poté slouží pro klasifikaci jen jednoho typu předmětů. Pokud bychom ve VHDL vytvořili jeden konkrétní natrénovaný klasifikátor, sloužil by jen pro jednu činnost a nedal by se snadno přetvořit na jinou klasifikační úlohu. Například klasifikátor naučený na rozpoznávání obličejů by šlo jen těžko přetvořit na klasifikátor pro rozpoznávání aut.

Jednou z možností pro řešení tohoto problému, kterou VHDL nabízí, je napsat generický kód ve VHDL, který by se měnil na základě zadaných generických parametrů. Takové řešení je velmi komplikované pro realizaci ve VHDL a nelze zaručit, že by řešení šlo vytvořit pro všechny možné typy klasifikátorů. Dále pak nelze zajistit, aby řešení bylo jednoduše rozšířitelné o další klasifikátory. Jedním z problémů, který by se musel řešit, je inicializace vyhledávacích tabulek (LUT). Jedná se o vypsání paměťové reprezentace o velikosti až několika desítek kilo bytů.

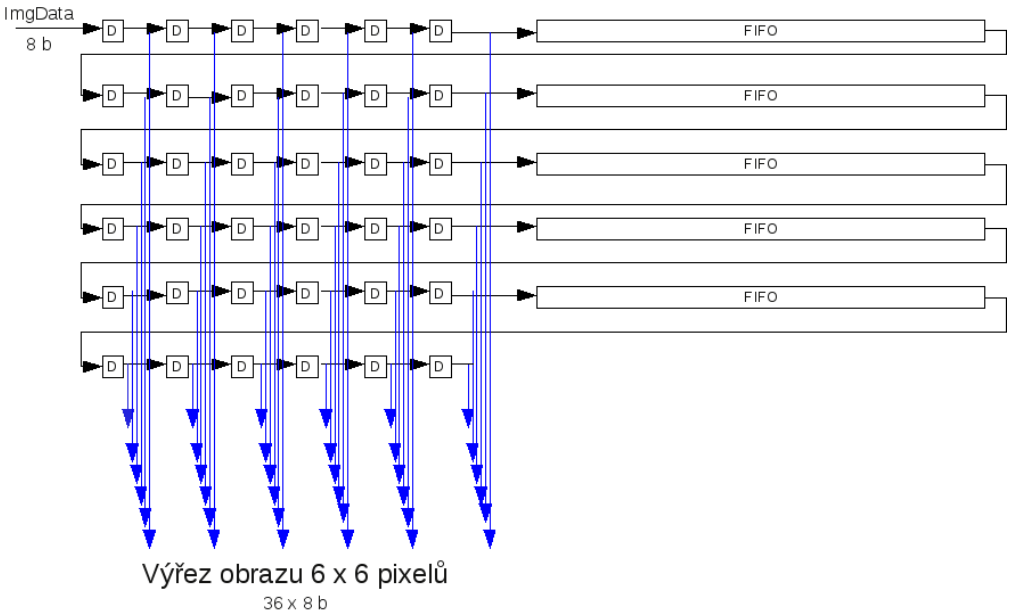
Druhý možný způsob je generování VHDL kódu z univerzálního programovacího jazyka, jakým je například C++. Tato možnost dává velmi široké možnosti, jak architekturu vygenerovat. Při tomto přístupu je přenesena podstata generických parametrů z VHDL do jazyka C++. Tímto způsobem by nebyly potřeba generické parametry ve VHDL. Vše by se vygenerovalo z vyššího programovacího jazyka. Nevýhodou je, že by se museli generovat všechny soubory architektury ve VHDL. A každý generovaný soubor lze poté jen těžko později upravovat. Při každé malé úpravě souboru by se musel udělat zásah do zdrojových souborů programu generujícího VHDL kód. V případě jazyka C++ by se musel program navíc znovu zkompilovat.

Jako nejlepší řešení se jeví spojení obou dříve uvedených způsobů dohromady. A tedy že jednotlivé komponenty ve VHDL budou mít generické parametry a zároveň se vybrané komponenty budou generovat z VHDL kódu. Takto můžeme některé komponenty napsat jako generické ve VHDL a nemusí se generovat z vyššího programovacího jazyka. Měnící se komponenty se naopak generovat budou. Tento způsob byl zvolen pro implementaci generátoru VHDL kódu.

V následujících kapitolách bude popsána architektura na úrovni jednotlivých komponent, ze kterých se skládá výsledný systém (dle obrázku 26).

5.1 ImgBuffer

Komponenta *ImgBuffer* slouží k uchování části obrazu, na kterém probíhá klasifikace. Pro realizovaný klasifikátor s klasifikačním oknem o velikosti *6 x 6* pixelů, je nutné ukládat pět kompletních řádků obrazu a šest pixelů z šestého řádku. V každém kroku (taktu) vstupuje do jednotky jeden pixel obrazu. Zároveň v každém kroku z jednotky vystupuje třicet šest hodnot pixelů představujících klasifikační okno o velikosti *6 x 6* pixelů.



Obrázek 29: Ukládání dat v komponentě ImgBuffer.

Na obrázku 29 je ukázán způsob uložení dat. Vždy se ukládá pruh obrazu. Pokud pixel projde všemi zpožďovacími linkami, je jeho hodnota zahozena (nikdy se s ním již nepracuje). Pro uložení dat výřezu obrazu o velikosti *6 x 6* pixelů se používají zpožďovací obvody, které vytváří zpoždění o velikosti jeden takt (D klopný obvod). Výstup každého takovéhoto paměťového členu je přiveden na výstupní sběrnici.

Pomocí zpožďovacích paměťových členů se dosáhne uložení 6 pixelů v každém řádku. Pro uložení zbývajících pixelů v řádku se používá FIFO[10] zpožďovací linka. Například při šířce obrazu 512 pixelu se bude šest pixelů ukládat v paměťových členech a zbylých 506 pixelů se uloží do zpožďovací FIFO linky. FIFO linka zajišťuje, že se pixely z aktuálně zpracovávaných řádků neztrácí, ale jsou uloženy. Ve správný čas se objeví jako vstupní hodnota pro následující řádek, který má opět stejnou strukturu. Tedy šest zpožďovacích obvodů následovaných FIFO linkou.

FIFO linka je obvykle velká a není vhodné ji vytvořit pomocí samostatných paměťových prvků, ale je vhodnější použít blokovou pamět RAM (*BlockRAM*), která je dostupná na FPGA čipu. FIFO linka se z blokové paměti vyrobí za použití čítače, jako adresovacího prvku.

Počet zdrojů, které komponenta pro ukládání obrazu zabírá na čipu, je závislý na šířce vstupního obrazu. Část obvodu pro ukládání a zpřístupňování výřezu obrazu o velikosti *6 x 6* pixelů musí mít vždy stejný počet registrů pro ukládání dat. Ta se nemění. Co se však se šířkou obrazu měnit musí je reprezentace FIFO zpožďovacích linek. Pro malou šířku obrazu by bylo možné reprezentovat všechny zpožďovací linky pomocí jedné *BlockRAM* paměti na čipu FPGA, ve spojení s čítačem, který slouží jako adresovací logika pro vytvoření FIFO linky z *BlockRAM* paměti.

Paměť *BlockRAM* lze nakonfigurovat na maximální šířku vstupu 2 x 36 bitů. Při této konfiguraci je k dispozici 512 adresovatelných paměťových míst. Zde je nutné si uvědomit, že *BlockRAM* má dva vstupní a dva výstupní porty, které jsou navzájem nezávislé. Jediným omezením pro jejich používání je, že nelze přistupovat najednou oběma porty na stejné paměťové místo, přičemž alespoň jedna z operací je zápisová.

[10] FIFO – First In Firs Out – první dovnitř, první ven.

Maximální šířka obrazu	Konfigurace BRAM	Počet BRAM	Využití BRAM
262 pixelů	2 x 36 bitů	1	55,5%
518 pixelů	1 x 36 bitů	2	55,5%
1030 pixelů	1 x 16 bitů	3	83,3%

Tabulka 1: Odhad paměťové náročnosti komponenty ImgBuffer.

Tabulka 1 ukazuje počet *BlockRAM* modulů potřebných pro vytvoření FIFO linek v závislosti na šířce obrazu. První sloupec udává maximální šířku obrazu. Pokud bude vstupní obraz užší, než uvedená mez, v návrhu se neušetří žádné zdroje. Jen část paměti *BlockRAM* bude nevyužita. To je zapříčiněno možnostmi adresování paměťových míst v *BlockRAM*. Tuto paměť lze nakonfigurovat do několika módů. U každého řádku tabulky je uvedeno, v jaké konfiguraci se paměť *BlockRAM* použije. Velmi zajímavým ukazatelem je využití paměti *BlockRAM*. Pro obraz maximální šířky 262 pixelů se použije jen jedna *BlockRAM* paměť a její využití je jen 55 procent. Podobně je tomu i u maximální šířky obrazu 518 pixelů. Zlom nastává při maximální šířce obrazu 1030 pixelů. V tomto případě je již využití *BlockRAM* paměti na 83 procentech.

Maximální šířka obrazu 262 pixelů je pro zpracování obrazu příliš malá a tak se s ní v dalším návrhu nebude pracovat. Za dostatečnou šířku obrazu lze považovat hodnotu 512 pixelů. Pokud bude vstupní obraz větší než 512 pixelů, můžeme jej rozdělit na dvě části a každou zpracovat samostatně. Rozdělení na dvě části však sebou nese povinnost vytvořit při rozdělení obrazu na dvě části jejich vzájemný překryv. Velikost překrytí je dána šířkou detekčního okna. Do FPGA poté musí jít obrázek, který je rozdělen na dvě poloviny. Každá část by se zpracovala samostatně. Rozdělení obrazu na dvě nebo více částí by obstarával buď počítač, nebo obslužný DSP[11] procesor.

5.2 Komponenta SUM

Komponenta *SUM* slouží pro výpočet konvolucí nad obrazem. Každá konvoluce je dána svou pozicí a tvarem. Tvary konvolucí jsou určeny konfiguračním souborem a dají se dle požadavků klasifikačních úloh měnit. Podmínkou pro typ konvoluce je, že počet pixelů, které se v ní uplatní, je roven mocnině dvou. Tento požadavek je kladen z důvodu normalizace výsledku konvoluce do osmibitového čísla. Normalizace se provádí pomocí celočíselného dělení. Jelikož je pro velikost konvoluce zadáno pravidlo, které popisuje její velikost jako mocninu dvou, víme, že se vždy bude provádět dělení, které lze převést na logický posuv vpravo. Hodnotu, o kterou se provede logický posuv vpravo, lze vypočítat jako $\log_2 N$, kde N představuje počet pixelu, které se uplatní v konvoluci.

Operace logického posuvu vpravo lze v FPGA provést velmi jednoduše. Dokonce se při její realizaci nespotřebují žádné zdroje. Provede se tak, že se vynechá LSB[12] bit. V případě dělení větším číslem se vynechá více bitů sousedících s LSB.

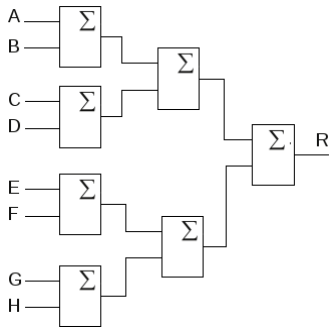
Jelikož je množství konvolucí velké, není vhodné do cílového návrhu umístit všechny možné typy konvolucí. Proto je blok SUM vhodným kandidátem na generování z vyššího programovacího jazyka. V procesu generování je možné vygenerovat jen nezbytné množství konvolucí. Zároveň je možné provádět základní optimalizace při výpočtu konvolucí.

[11] DSP – Digital Signal Processor

[12] LSB - Least Significant Bit – nejméně významný bit

Sčítačky na čipu FPGA jsou dvou vstupové. To znamená, že v jednom kroku můžeme sečíst jen dva bity signálu, případně dva vstupní vektory. Pokud bychom chtěli sečíst tři hodnoty, musíme nejprve provést součet dvou hodnot a poté výsledek součtu sečíst s třetí vstupní hodnotou.

Jelikož je možné sčítat jen dvě hodnoty v jednom kroku, vede sčítání více hodnot na kaskádu sčítaček. Nejefektivnější je sčítat vstupní hodnoty po dvojicích. Výsledky součtu z předchozího kroku opět sečíst po dvojicích a tak pokračovat až k získání poslední, cílové, hodnoty.



Obrázek 30: Kaskáda jednobitových sčítaček.

Na obrázku 30 je znázorněna kaskáda jednobitových sčítaček (pro jednoduchost se neuvažuje přenos při sčítání - *carry*). Při návrhu FPGA nás zajímá rychlost, jakou daný obvod poběží. Pokud uděláme v FPGA příliš velkou kaskádu sčítaček, bude tento obvod příliš pomalý a bude „brzdit“ zbytek obvodu. Stupeň kaskády získáme jako dvojkový logaritmus počtu vstupů ($\log_2 N$, kde N je počet vstupů).

Obvykle bývá v jednom návrhu klasifikátoru třeba počítat několik druhů konvolucí. V návrhu se tak objeví několik různě velkých sčítacích kaskád. Pro správnou činnost klasifikátoru potřebujeme mít na výstupu bloku SUM všechny konvoluce současně. To však klade požadavek, že různě dlouhé kaskády musí produkovat svůj výstup ve stejném čase (synchronně). Tento požadavek je z technologického hlediska nesplnitelný (pokud nepoužijeme zpožďovací prvky).

Současného výstupu všech bloků tak můžeme dosáhnout pomocí rozdělení sčítací kaskády do několika stupňů. Zavedeme zde takzvaný vnitřní pipelining. Tímto krokem dosáhneme potřebné rychlosti kaskády sčítaček. A všechny kaskády budou produkovat svůj výstup synchronně. Avšak zvětší se latence (odezva) obvodu. Ta ale není podstatná, jelikož zpracováváme proud vstupních dat.

Uvedeným řešením pomocí pipeliningu jednotlivých stupňů dosáhneme zvýšení rychlosti výpočtu. Zároveň však vytvoříme problém se synchronizací dat na výstupu bloku *SUM*, při současném běhu více různých sčítacích kaskád. Pro vyřešení tohoto problému je nutné u sčítacích kaskád, které jsou kratší než nejdelší použitá sčítací kaskáda, použít výstupní zpožďovací linku o požadované délce.

Za předpokladu, že postačující rychlost návrhu v FPGA je 100 MHz, nemusíme vkládat zpoždění za každý stupeň sčítací kaskády, ale až za každý druhý stupeň kaskády.

Při generování komponenty *SUM* se generují výše uvedené sčítací kaskády. Jak již bylo dříve napsáno, obvod je sestaven jen z dvou-vstupových sčítaček. Pří generování dochází ke kontrole, zda právě generovaná sčítačka již není v návrhu vygenerována. Pokud je již jednou vygenerována, použije se již vygenerovaná sčítačka. Tímto je prováděna optimalizace obvodu již při generování VHDL kódu a dochází k minimalizaci potřebných zdrojů.

5.3 Slabé klasifikátory - příznaky

Výstup bloku *SUM* je uspořádán do devític po osmi bitech. Každá taková devítice je přivedena na vstup jedné nebo více jednotek vypočítávajících příznaky. Výstup každé jednoty vypočítávající příznaky je přiveden na vstup jedné nebo více *LUT*.

Slabé klasifikátory (příznaky) jsou nejdůležitější komponentou pro výpočet výsledného silného klasifikátoru. Všechny ostatní komponenty slouží jen pro přípravu vstupních dat pro slabé klasifikátory, nebo zpracovávají výstup slabých klasifikátorů.

Na všechny příznaky, které se mohou v architektuře vyskytnout, jsou kladeny stejné požadavky z hlediska vstupů, výstupů a časové náročnosti výpočtu. Tento požadavek je důležitý z hlediska možnosti záměny jednoho typu příznaku za jiný typ příznaku. V architektuře je dovoleno využívat následujících typů slabých klasifikátorů:

- LBP
- LRP
- LR

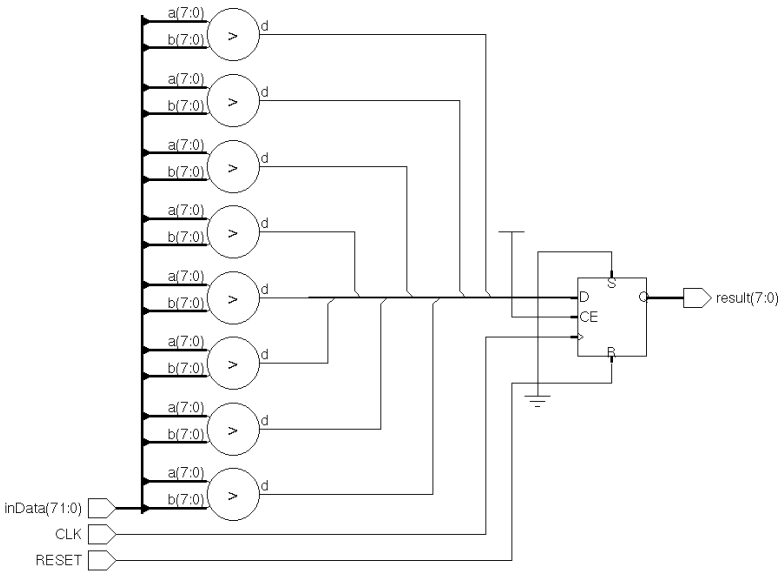
Všechny uvedené příznaky mají na vstupu devítici osmibitových hodnot, na výstupu mají jen jednu osmibitovou hodnotu. Zpracování každého příznaku trvá jeden takt.

Jelikož se jedná o neměnné jednotky, není třeba je generovat pomocí generátoru VHDL kódu. Výpočet LBP je vždy stejný, tak pro něj není nutné zavádět specifické generické parametry. U LRP příznaku je však situace jiná. V kapitole 2.5.3 je uveden způsob výpočtu LRP. Ve výpočtu tohoto příznaku vystupují dvě proměnné. První je pozice *ranku A* a druhou proměnnou je pozice *ranku B*. Tyto parametry je vhodné nastavit pomocí generických parametrů. A každou instanci LRP vygenerovat dle aktuálních parametrů příznaku. Stejná situace nastává i u LR příznaků.

LR příznak, na rozdíl od LRP a LBP, reprezentuje výstupní hodnotu jen na čtyřech bitech. Tento požadavek však není na obtíž, jelikož výstup lze bez problému rozšířit na požadovaný počet osmi bitů. Rozšíření může proběhnout například nulami. U LR příznaku je stejně jako u LRP použit generický parametr, pro určení pozice *ranku*, který udává konvoluci, se kterou se všechny ostatní hodnoty konvolucí porovnávají.

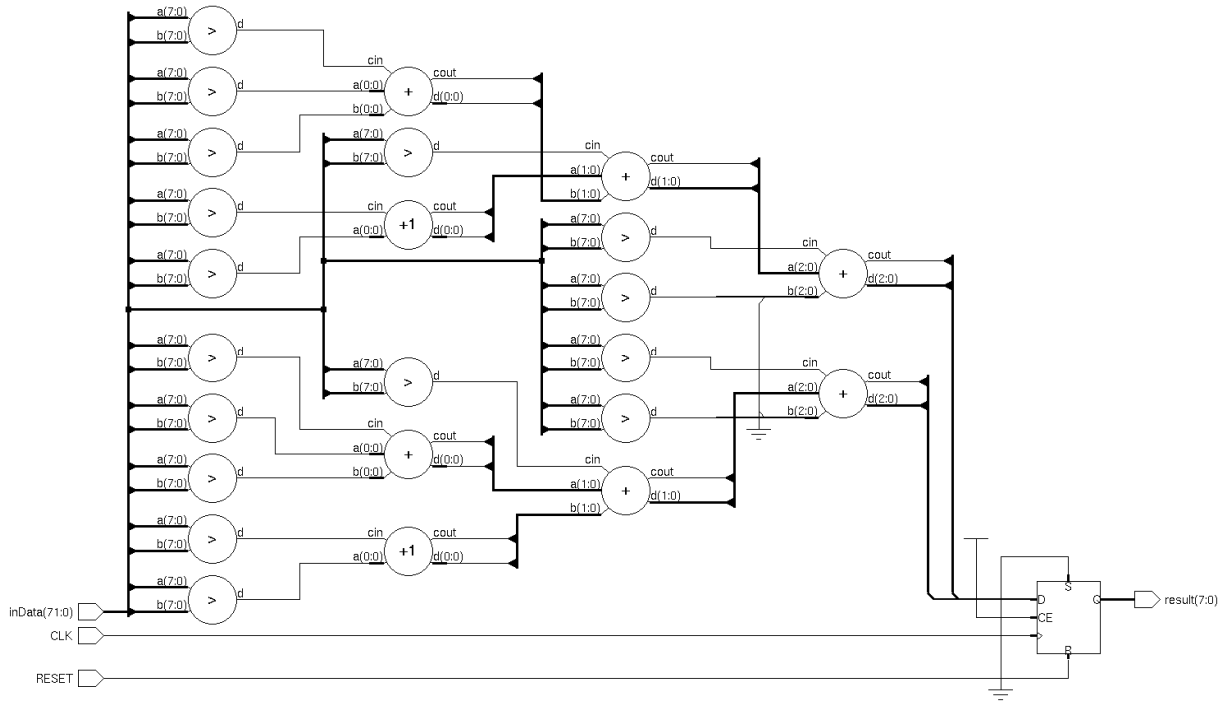
5.3.1 Local Binary Pattern (LBP)

Na obrázku 31 je uvedeno RTL schéma LBP příznaku. Obvod se skládá z několika komparátorů a výstupního registru. Výstup každého komparátoru tvoří jeden bit výstupního vektoru bitů. Je vhodný pro realizaci v FPGA a nezabírá mnoho zdrojů. Pro jeho realizaci je potřeba jen 32 *Slices*. Obvod může běžet při rychlosti 157,8 MHz. Uvedené údaje jsou jen orientační. Odhad náročnosti na zdroje a rychlost byl proveden pro FPGA rodiny Spartan3.



Obrázek 31: RTL[13] schéma LBP.

5.3.2 Local Rank Patterns (LRP)



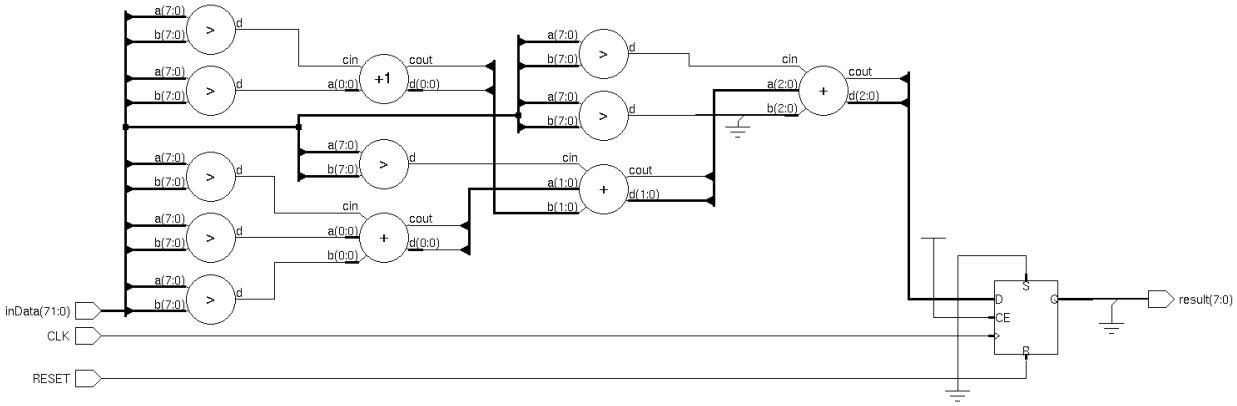
Obrázek 32: RTL schéma LRP.

Na obrázku 32 je znázorněno RTL schéma LRP příznaku. V porovnání s LBP klasifikátorem lze vidět, že obvod je asi dvakrát náročnější na realizaci v FPGA než dříve uvedený LBP příznak. Vygenerovaný obvod potřebuje pro svou funkčnost 87 *Slices*, což je více něž dvojnásobek, který potřebuje pro svoji realizaci LBP. LRP má větší kaskádu vzájemně propojených prvků. Proto může

[13] RTL – Register Transfer Logic – popis na úrovni registrových přenosů

pracovat na maximální frekvenci jen 69,2 MHz. Řešením jak urychlit tento obvod je jeho rozdělení na dvě části a vložení registrů mezi tyto dvě části. Touto úpravou se zvýší pracovní frekvence obvodu nad hranici přesahující 100 MHz.

5.3.3 Local Rank (LR)



Obrázek 33: RTL schéma LR klasifikátoru.

Na obrázku 33 je uvedeno RTL schéma LR příznaku. LR koncepčně vychází z LRP. LR příznak implementuje jen polovinu příznaku LRP. Z tohoto důvodu zabírá poloviční množství zdrojů (39 *Slices*). Avšak rychlost obvodu je porovnatelná s LRP klasifikátorem (78,3 MHz). Možnost zrychlení obvodu opět spočívá v jeho rozdělení na více částí, jako bylo uvedeno u LRP příznaku.

5.3.4 Porovnání příznaků

V předešlých kapitolách byly představeny jednotlivé příznaky, které se mohou v návrhu použít. Nejlepší příznak z hlediska implementace v FPGA je LBP. Zabírá nejméně místa na čipu a zároveň je ze všech uvedených nejrychlejší. Silné klasifikátory, které jsou výstupem trénování, zpravidla používají jen jeden typ slabého klasifikátoru. Navržená architektura umožňuje použití všech druhů slabých klasifikátorů v jednom návrhu.

Typ klasifikátoru	Zdroje	Rychlost
LBP	32 Slices	157,8 MHz
LRP	87 Slices	69,2 MHz
LR	39 Slices	78,3 MHz

Tabulka 2: Požadavky klasifikátorů na zdroje.

V tabulce 2 je uveden přehled náročnosti jednotlivých příznaků v závislosti na použitých zdrojích. Tabulka ukazuje, kolik každý příznak zabere zdrojů a jak bude rychlý. Ale nic neříká o tom, jak bude vypadat architektura pro jeden natrénovaný silný klasifikátor. Pro jednoduchost předpokládejme, že máme natrénované tři silné klasifikátory. V každém klasifikátoru je vždy použit jen jeden typ slabého klasifikátoru. Dále předpokládejme, že všechny klasifikátory používají čtyři druhy konvolucí. Dalším předpokladem je, že všechny silné klasifikátory jsou složeny z dvaceti slabých klasifikátorů.

Vstupem do všech slabých klasifikátorů jsou tedy čtyři devítice osmibitových hodnot. V případě klasifikátoru LBP musíme vložit do architektury čtyři instance LBP příznaků. Tyto čtyři instance se přivedou na vstup dvaceti LUT. Tímto získáme požadovaný počet dvaceti slabých klasifikátorů.

Pokud budeme uvažovat silný klasifikátor složený jen z LRP nebo jen z LR klasifikátorů, může jich být v cílové architektuře vytvořeno mnohem více, než v klasifikátoru složeném jen z LBP příznaků. Je to dáno tím, že každý klasifikátor má na vstupu kromě konvoluce i pozici *rank A* a *rank B*. Kvůli těmto parametrům může mít každý slabý klasifikátor jinou konfiguraci a bude ve výsledné realizaci obvodu vložen vícekrát. V nejhorším případě může být LRP nebo LR příznak vložen do obvodu dvacetkrát (pro uváděný příklad).

Proto může být klasifikátor tvořený LRP nebo LR slabými klasifikátory mnohem náročnější na počet zdrojů, které spotřebuje, než klasifikátor složený jen z LBP slabých klasifikátorů. Další nevýhoda při použití LRP příznaků nastává při generovaní *LUT* obvodů (viz následující kapitola).

5.4 LUT – Vyhledávací tabulky

LUT - neboli vyhledávací tabulky, představují data, které jsou výstupem trénovacího procesu. Pro každý slabý klasifikátor je k dispozici jedna *LUT*. Výstup LBP klasifikátoru může nabývat 256 různých hodnot. LRP klasifikátor může mít na výstupu celkem 81 různých hodnot. LR klasifikátor si vystačí pouze s 9 různými hodnotami výstupu. Těmto hodnotám odpovídají velikosti *LUT* tabulek pro každý typ klasifikátoru.

Hodnoty, které tvoří *LUT*, jsou nazývány jako takzvané alfa koeficienty. Po celý proces klasifikace se na ně pohlíží jako na konstanty. Hodnoty je nutné uložit na čipu FPGA, jelikož se sněmy pracuje velmi často. Jejich uložení v externí paměti RAM tak není příliš vhodné, vzhledem k rychlosti přístupu k takto uloženým datům. Jedním ze způsobů, jak vyhledávací tabulky uložit je využití pamětí *BlockRAM*. Jiným možným způsobem je vytvoření distribuované ROM[14] paměti. Distribuovaná ROM paměť se vytváří pomocí spojování jednotlivých Slices a zabere tak zdroje na čipu FPGA.

Řešení s využitím distribuované paměti ROM je vhodné jen pro malé paměti o kapacitě několika bytů. Z hlediska adresování je možné vytvářet paměti, které mají velikost rovnu mocnině o základu dvou. Tento požadavek vyplývá ze způsobu adresování paměti. Pro vytváření *LUT* je vhodné použít *BlockRAM* paměti. Jedna *BlockRAM* paměť má velikost 18 432 bitů. Jedna *LUT* o velikosti 256 prvků potřebuje pro své uložení 2048 bitů. Do jedné paměti *BlockRAM* lze tak teoreticky uložit 9 *LUT*. Takové uložení *LUT* je příliš idealizováno a může nastat jen za velmi specifických podmínek.

Problémem, který není dobře řešitelný je ten, že potřebujeme v každém taktu z každé vyhledávací tabulky *LUT* vyčítat hodnotu uloženou na požadované adrese. Adresa do každé *LUT* paměti se může lišit.

BlockRAM je možné využít v konfiguraci se dvěma nezávislými vstupy a výstupy - porty. Na každém portu tak lze číst hodnotu uloženou na požadovaném místě v paměti. Tato konfigurace *BlockRAM* nám však dovoluje využít jen 4096 bitů z cekem 18 432 bitů. Tím dostaneme využití *BlockRAM* paměti 22 procent. Takové využití je příliš nízké pro implementaci a na realizaci všech *LUT* tabulek bychom spotřebovali příliš mnoho *BlockRAM* paměťových modulů.

Jistá optimalizace se ukazuje, pokud se bude v rámci silného klasifikátoru vyskytovat více slabých klasifikátorů, které budou mít stejný vstup (stejný vstup z bloku *SUM*), konfiguraci (LBP, LRP se stejnými ranky atd.) a budou se lišit pouze pozicí v rámci detekčního okna. Samozřejmostí je,

[14] ROM – Read Only Memory – pamět, kterou lze pouze číst

že každý klasifikátor bude mít jiné hodnoty v *LUT*. Tento případ se vyskytuje, zejména pokud je silný klasifikátor tvořen LBP příznaky. V takovém případě může nastat, že je vygenerován potřebný počet LBP klasifikátorů a na jejich výstupy je napojeno 20 *LUT* (přepokládejme silný klasifikátor, jež je tvořen 20 slabými klasifikátory). Pokud by se neprováděla žádná optimalizace, spotřebovalo by se na realizaci 10 *BlockRAM* pamětí. Optimalizací jsme schopni tento počet snížit. Předpokládejme, že jednotlivým instancím LBP odpovídá zadaný počet *LUT* tabulek:

- LBP_1 … 9 *LUT* pamětí
- LBP_2 … 5 *LUT* pamětí
- LBP_3 … 4 *LUT* paměti
- LBP_4 … 2 *LUT* paměti

Celkový počet *LUT* je 20. *BlockRAM* dovoluje nakonfigurovat každý svůj port na šířku vstupu a výstupu 36 bitů. Při možnosti adresování 512 paměťových míst. Každý port tak může obsluhovat nezávisle na druhém, 256 paměťových míst. Při šířce dat 2 x 36 bitů dostáváme celkovou šířku vstupu a výstupu 72 bitů. Tato šířka odpovídá 9 hodnotám z LUT. Najednou je tak možné vyčíst z paměti 9 hodnot. Je nutné zde poznamenat, že část paměti *BlockRAM*, která je určená pro ukládání paritních bitů, využíváme pro uložení dat *LUT*. Devět hodnot z vyhledávacích tabulek *LUT* je však možné najednou vyčítat z paměti, jen pokud je adresa do všech *LUT*, realizovaných pomocí jednoho modulu *BlockRAM*, stejná.

Pokud se podíváme na zadané rozvržení *LUT* pamětí, vzhledem k jednotlivým LBP klasifikátorům, vidíme, že pro klasifikátor LBP_1 máme realizovat devět *LUT* pamětí. Můžeme je tedy realizovat pouze jediným modulem paměti *BlockRAM*. Jen u tohoto případu je úspora tři a půl *BlokRAM* paměti oproti realizaci bez optimalizace. Toto je však nejideálnější případ, kdy jsme schopni využít *BlockRAM* paměť na sto procent. K takovým konfiguracím silných klasifikátorů nedochází v reálném případě příliš často.

Abychom mohli využívat takto efektivně paměť, musí v ní být obsah jednotlivých *LUT* správně uložen. Hodnoty se do paměti *BlokRAM* musejí ukládat s rozestupem. Jelikož při každém čtení získáme na každém portu paměti *BlockRAM* třicet dva souvislých bitů paměti a čtyři bity z paritního prostoru, je nutné hodnoty *LUT* uložit v paměti s příslušným rozestupem. Pro tento případ je rozestup hodnot jedné *LUT* paměti třicet dva bitů. O vygenerování správného paměťového rozestupu se stará generátor VHDL kódu.

Nyní se vraťme k zadanému příkladu. LBP_1 je již zpracován a v seznamu zůstali následující položky:

- LBP_2 … 5 *LUT* pamětí
- LBP_3 … 4 *LUT* paměti
- LBP_4 … 2 *LUT* paměti

Jako další se vybere ze seznamu klasifikátor LBP_2. Jelikož do každé poloviny *BlockRAM* paměti je možné uložit 4 *LUT*, pro uložení 5 *LUT* musíme použít obě poloviny *BlockRAM* paměti. Jedna polovina bude využita naplno a z druhé poloviny se využije jen jedna čtvrtina. Po této operaci se seznam zredukuje na položky:

- LBP_3 … 4 *LUT* paměti
- LBP_4 … 2 *LUT* paměti

Obě skupiny potřebují méně nebo rovno čtyřem jednotkám *LUT*. Proto můžeme obě skupiny realizovat pomocí jediné paměti *BlockRAM*, kdy každá skupina využije jednu polovinu paměťového modulu. Po tomto kroku jsou již všechny skupiny zpracovány.

Tímto jednoduchým optimalizačním způsobem jsme dosáhli, že se pro realizaci zadaného klasifikátoru použijí tři *BlockRAM* paměti namísto původních deseti. To je úspora sedmi *BlockRAM* modulů. Je to velmi dobrá hodnota. Využití pamětí BlockRAM je v uvedeném případě 74 procent.

V následujícím rámečku je uveden algoritmus pro generování *LUT* (pomocí pseudokódu).

```
ListByType  = SeskupKlasifikátoryDleTypu();
     /* jednu skupinu tvoří stejné klasifikátory */
while ( NeníPrázdný( ListByType ) ) do
     Group = vyberNejvětšíSkupinu (ListByType );
     if ( size(Group) > 4 ) then
          vygenerujBRAM( Group );
          if ( size(Group) > 9 ) then
               odstraňJižVygenerovéLUT ( Group );
          else
               odstraňGroup ( ListByType, group  );
          end if;
    else
          PřipravPolovinuBRAM( Group );
          OdstraňGroup ( ListByType, Group  );
          Group = ZískenNejmenšíSkupinu ( ListByType );
          if (  size(Group) > 4 ) then
               VygenerujBRAM();
          else
               PřipravDruhouPolovinuBRAM( Group );
               VygenerujBRAM();
               OdstraňGroup ( ListByType, group  );
          End if;
    end if;
done;
```

Algoritmus pro generování LUT.

Algoritmus pro generování *LUT* je jednoduchý a názvy všech funkcí jsou voleny tak, aby bylo zřejmé, jakou činnost vykonávají. Proto není nutné jej podrobně analyzovat v textu.

Výstup *LUT* je připojen na vstup následující bloku, který představuje výstupní zpožďovací linku v kombinaci se sčítačkami. Jelikož jsou *LUT* tvořeny pomocí *BlockRAM* jsou velmi rychlé. Není nutné je proto optimalizovat vzhledem k rychlosti.

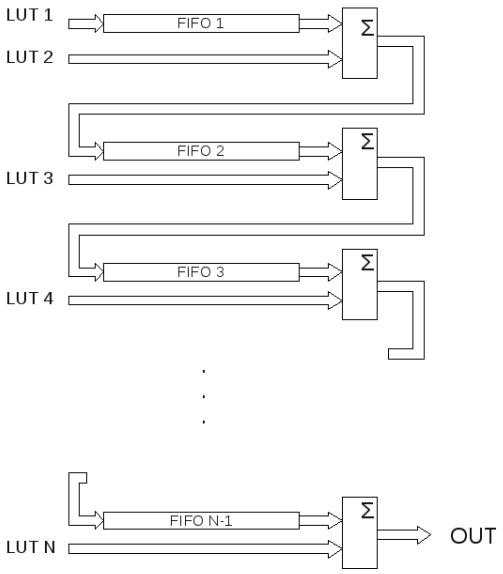
5.5 Výstupní zpožďovací linka - OutBuffer

Vstup do zpožďovací linky je tvořen výstupy *LUT* pamětí. Jejich počet je odvozen od počtu slabých klasifikátorů, které tvoří výsledný silný klasifikátor. V následující kapitole budeme předpokládat silný klasifikátor, který je tvořen dvaceti slabými klasifikátory. Vstupem do bloku *OutBuffer* je v uvedeném příkladě dvacet osmibitových hodnot. Výstupem je jedna šestnáctibitová hodnota. Výstup je šestnácti bitový, jelikož dochází ke sčítání několika (v uvedeném případě dvaceti) vstupních osmibitových hodnot. Dvacet vstupních hodnot, které vstupují současně, se v definovaných časech sčítá a vytvoří výstup. Součet dvaceti hodnot potřebuje pro svoji reprezentaci jen třináct bitů. Jelikož je však práce s třináctibitovým číslem nepohodlná, zavádí se mírná redundance a výstup se uvažuje jako šestnáctibitový. Potřebný počet bitů výstupního streamu dostaneme pomocí vzorce

$DATAWIDTH + \log_2 N$, kde *DATAWIDTH* je šířka vstupních dat a *N* je počet slabých klasifikátorů.

Pro každý natrénovaný silný klasifikátor je výstupní zpožďovací linka zpravidla odlišná. Proto je vhodné ji vytvářet pomocí generátoru VHDL kódu. Velikost výstupní linky je dána velikostí detekčního okna. Čím větší výšku bude mít detekční okno, tím více zdrojů se spotřebuje na vytvoření výstupní zpožďovací linky. Dalším parametrem, který významně ovlivňuje velikost výstupní zpožďovací linky, je šířka zpracovávaného obrazu. S rostoucí šířkou obrazu rostou velikosti jednotlivých meziřádkových zpoždění a tím i požadavky na zdroje.

Podrobný popis, jak funguje výstupní zpožďovací linka, je v kapitole 4.3.3. Nyní si uvedeme její schéma.



Obrázek 34: Schéma výstupní zpožďovací linky.

Na obrázku 34 je znázorněno schéma výstupní zpožďovací linky. Pro *N* vstupních hodnot se musí ve výstupní zpožďovací lince vytvořit *N-1* zpožďovacích linek a *N-1* sčítaček. Všechny sčítačky budou mít vždy stejnou velikost. Ale velikost jednotlivých FIFO linek se bude měnit v závislosti na parametrech slabých klasifikátorů. Konkrétněji, délka každé FIFO linky *k*, je dána jako rozdíl pozice slabého klasifikátoru *k+1* a *k* (pozice klasifikátorů se berou v rámci detekčního okna).

Realizace zpožďovacích FIFO linek je možná pomocí dvou různých přístupů. Prvním způsobem je vytvoření zpožďovací linky pomocí několika registrů. Druhou možností je použití *BlockRAM* pamětí ve spojení s čítačem. Tak je možné vytvořit kruhový buffer, který bude reprezentovat FIFO linku o zadané délce.

První z uvedených řešení je výhodné tam, kde je požadované malé zpoždění. Pojem „malé“ zpoždění můžeme definovat jako šířku detekčního okna (to je například 24 pixelů). Můžeme tak vygenerovat malé zpoždění bez použití *BlockRAM* paměti, jen s využitím několika *Slices* na čipu.

Pokud však zpoždění přesáhne stanovenou mez, je vhodnější použít FIFO linku vytvořenou pomocí *BlockRAM* pamětí. V návrhu se uvažuje šířka dat pro FIFO linku 16 bitů. Pro takovou šířku dat můžeme *BlockRAM* paměť použít v konfiguraci se vstupní šířkou 16 bitů pro každý port. Při této konfiguraci můžeme adresovat 1024 paměťových míst. Délka FIFO linky však může být mnohem delší než uvedená hodnota. Poté se musí spojit několik *BlockRAM* pamětí, aby se vytvořila jedna dlouhá zpožďovací linka.

Dalším případem je, pokud jsou délky linek větší než detekční okno, a tak je není vhodné reprezentovat pomocí několika registrů, a zároveň jejich délka je o hodně menší než je maximální velikost FIFA vytvořeného pomocí *BlockRAM* paměti. Nyní se nabízí možnost využití dvou portů u paměti *BlockRAM*. Poté jsme schopni do jedné paměti *BlockRAM* umístit dvě zpožďovací linky a zvýšit tak její využití. Pro optimální využití *BlockRAM* pamětí je ale nutné provést optimalizaci, která povede k vygenerování takové reprezentace FIFO linek.

Níže je uveden algoritmus, který zadaný problém řeší. V algoritmu zapsaném pomocí pseudokódu nejsou ošetřeny všechny stavy, které mohou nastat. Algoritmus by se tím neúměrně prodloužil. V pseudokódu algoritmu také není žádným způsobem řešeno mapování vstupních portů pro každou FIFO linku a generování pomocných funkcí. Tyto dodatečné věci však nejsou potřebné pro vysvětlení funkce generování výstupní zpožďovací linky.

Algoritmus pracuje na principu, že nejprve vygeneruje krátké zpožďovací linky (zpoždění) a následně dlouhé zpožďovací linky. Po vygenerování krátkých zpožďovacích linek, algoritmus zpracuje zbývající „dlouhé“ zpožďovací linky. Kdykoliv během generování dlouhých zpoždění narazí na krátké zpoždění, vygeneruje jej pomocí zpožďovací linky, která je tvořena registry.

Algoritmus vybere FIFO linku s největším zpožděním. Vybranou linku postupně zpracovává a generuje zpožďovací linky, ze kterých se bude skládat výsledná linka pro zpracovávané zpoždění. Aby se dosáhlo maximálního využití *BlockRAM* pamětí. Generují se do jedné paměti *BlockRAM*, pokud je to možné, dvě zpožďovací linky. Na každý port *BlockRAM* se generuje jedno zpoždění. Ideou celého řešení je vygenerovat do jedné *BlockRAM* paměti dvě zpoždění, kdy jedno zpoždění je zpravidla delší a druhé kratší. Zároveň se požaduje maximální využití *BlockRAM* paměti. FIFO linky je možné libovolně dělit na několik částí. Díky tomu je možné uplatnit následující strategii. Ze seznamu nezpracovaných linek se vybere linka s nejkratším zpožděním. Pokud je velikost nejkratší linky větší než je maximální délka FIFO linky vytvořitelné pomocí jedné *BlockRAM* paměti, tak nejkratší linku dále neuvažuju a vygeneruje se jen FIFO linka s největším zpožděním.

Pokud je však nejkratší FIFO linka menší než nejdelší možné zpoždění, jež je možné vygenerovat pomocí jedné *BlockRAM*, tak se na jeden port se vygeneruje kratší zpoždění a na druhý port se do zbylého místa vygeneruje co největší část nejdelší FIFO linky. Po vygenerování se upraví zbylá požadovaná délka nejdelšího FIFA a algoritmus se opakuje.

Algoritmus pokračuje, dokud nevygeneruje všechny požadované FIFO linky. V celém procesu generování je třeba vytvářet pomocné signály na propojení jednotlivých fyzických částí, ze kterých se jedna kompletní FIFO linka skládá.

```
DelaysList  = VypočtiZpoždění(); /* Vypočet všech zpoždění */
for d in DelaysList do  /* vygenerování krátkých zpoždění */
  if ( d.delkaZpoždění < šířkaDetekčníhoOkna  ) then
    generateRegFIFO( d );
    odstraň( DelaysList, d );
  end if;
done;
while ( NeníPrázdný( DelaysList ) ) do
  maxDelay = ZískejNejvětšíZpoždění ( DelaysList );
  while ( true ) do
    if ( size ( DelaysList ) > 1 ) then
      minDelay = ZískejNejmenší zpoždění ( DelaysList );
      if ( minDelay.delkaZpoždění < MaximálníDéklaFIFA ) then
        if ( maxDelay.delkaZpoždění < šířkaDetekčníhoOkna  ) then
          generateRegFIFO( maxDelay ); /* generování krátkých zpoždění*/
          odstraň( DelaysList, maxDelay );
          break;
        end if;
        if  ( MaximálníDéklaFIFA - minDelay.delkaZpoždění <
                                   maxDelay.delkaZpoždění ) then
          cntA = MaximálníDéklaFIFA - minDelay.delkaZpoždění;
        else
          cntA = maxDelay.delkaZpoždění;
        end if;
        generuj2PortFIFO ( cntA, minDelay );
        if ( cntA > 0 ) then
          odstraň ( DelaysList, minDelay );
        end if;
        if (cntA == maxDelay.delkaZpoždění ) then
          odstraň( DelaysList, maxDelay );
          break;
        else
          maxDelay.delkaZpoždění = maxDelay.delkaZpoždění – cntA;
        end if;
      else /* minimum je příliš velké, generuje se jen maxDelay*/
        GenerujBRAMFIFO ( maxDelay );
        if ( maxDelay.delkaZpoždění – MaximálníDéklaFIFA <= 0 ) then
          odstraň ( DelaysList, maxDelay );
          break;
        else
          maxDelay.delkaZpoždění = maxDelay.delkaZpoždění –
                                       MaximálníDéklaFIFA;
        end if;
      end if;
    else /* V seznamu je jen poslední prvek */
      vygenerujPosledníFIFO ( maxDelay );
      odstraň ( DelaysList, maxDelay );
    end if;
  done;
done;
```

Algoritmus pro generování výstupní zpožďovací linky.

5.6 Top level architektura

Již byla popsána kompletní architektura, ale nic nebylo napsáno o jejím rozhraní. Nezbytným vstupem do systému je hodinový signál – *CLK* a signál pro restartování obvodu *RESET*. Vstup dat do klasifikátoru tvoří osmibitový stream dat. Data do obvodu vstupují každý hodinový cyklus. Stejně tak i každý hodinový cyklus data vystupují z obvodu. Výstup obvodu je tvořen šestnácti bitovým streamem dat. Celý obvod pracuje synchronně. Funkce obvodu se dá pozastavit pomocí takzvaného povolovacího vstupu *EN*. Pokud je vstup aktivní, tak obvod pracuje a každý tak produkuje výsledky klasifikace. Pokud je neaktivní, obvod zastaví svoji činnost a na výstupu bude poslední zpracovaná hodnota. V neaktivním stavu obvod nepřijímá žádné data na svém vstupu. Každá z jednotek, ze kterých se obvod skládá, generuje vlastní zpoždění. Obvod je tak tvořen pomocí několika stupňů pipeline.

Název obvodu	**Zpoždění**
ImgBuffer	1CLK
SUM	1CLK – 4 CLK (dle použitých konvolucí)
LBP, LRP, LR	1 CLK nebo 2 CLK
LUT	1 CLK
OutBuffer	1 CLK
Celkem	**5** CLK – 9 CLK

Tabulka 3: Stupně pipeliningu v architektuře.

V tabulce 3 je uvedeno jakého zpoždění se dosahuje v každém výpočetním bloku. Tato zpoždění v součtu vyjadřují celkový pipelining obvodu. Jelikož je obvod generovaný z vyššího programovacího jazyka a každý silný klasifikátor má jiné požadavky, může se stupeň pipeliningu měnit. Největší změny ve stupni pipeliningu se mohou vyskytnout v bloku *SUM*. Stupeň pipeliningu je závislý na požadovaných konvolucích (viz 5.2).

V popisu architektury *top level* designu jsou umístěny všechny komponenty, ze kterých se návrh skládá. Komponenty *SUM*, *OutBuffer* a *ImgBuffer* jsou celé umístěny ve zvláštních souborech a do *top level* designu jsou vloženy jako samostatné komponenty. Komponenty pro výpočet příznaků a *LUT* jsou také navrženy jako samostatné komponenty. Ale jelikož počet instancí jednotlivých komponent je závislí na vstupních konfiguračních datech, musejí se do návrhu vkládat vícekrát. Jejich umístění probíhá vždy do architektury top level designu.

5.7 Generátor VHDL

Generátor VHDL kódu byl již po částech představen. Nyní zbývá jen doplnit, co potřebuje VHDL generátor pro svůj běh. Jsou to následující soubory:

1. Popis silného klasifikátoru.
2. Popis konvolucí, které se mohou vyskytnout při klasifikaci.
3. Konfigurační soubor.

Popis silného klasifikátoru vstupuje do procesu zpracování jako XML[15] soubor. V XML souboru jsou popsány všechny slabé klasifikátory, z nichž se výsledný silný klasifikátor skládá. V pořadí druhým vstupním souborem je popis konvolucí. Tento soubor není v textové formě, ale je napsán ve formě hlavičkového souboru jazyka C. Soubor se při překladu přikompiluje ke generátoru VHDL kódu. Důvodem, proč je tento soubor uváděn, jako hlavičkový, je jeho snadná přenositelnost, mezi trénovacím procesem a procesem generování. Oba soubory nebyly navrženy autorem této práce, ale jen použity. Jak již bylo zmíněno v předchozích kapitolách, práce se nezabývá trénováním klasifikátorů, ale využívá již natrénovaných klasifikátorů, které vznikly jako výsledek práce jiných pracovníků [14], [16].

Výstupem generátoru je adresář, který obsahuje již kompletně sestavený klasifikátor. V souborech je například vygenerován obsah pamětí ROM pro *LUT*. Celý návrh klasifikátoru obsahuje mnoho souborů a v nich napsaných entit. Ne všechny entity se však musí nutně generovat. Z toho důvodu se vybrané soubory negenerují, ale jen se kopírují do cílového adresáře. Soubory, které se pouze kopírují, obsahují generické parametry, pomocí nichž se nakonfigurují vkládané entity na požadovanou činnost. Soubory, které jsou výsledkem generování, se taktéž umístí do cílového adresáře.

Třetím souborem, který je potřebný pro běh generátoru, je konfigurační soubor. Tento soubor obsahuje například seznam souborů, které se mají do výsledného návrhu jen kopírovat, datové šířky použitých sběrnic a další podobné implementační nastavení. Pomocí konfiguračního souboru se také nastavují generické parametry architektur.

[15] XML – eXtended Markup Language

6 Testování a simulace

Testování architektury může probíhat dle dvou různých strategií. První možností je testování správné funkčnosti jedné konkrétní vygenerované varianty. Při této možnosti se ověřuje především správnost implementace. Druhou možností pro testování je generování různých architektur klasifikátorů a porovnávání jejich vlastností s ohledem na použité zdroje na čipu FPGA a výslednou rychlost návrhu. Každá strategie ukáže vlastnosti návrhu z jiného pohledu. Zatímco první strategie je především jen ověření předpokládaných vlastností, výsledkem druhé uvedené strategie může být návrh na optimalizace v návrhu.

6.1 Simulace vybrané architektury

Simulace vybrané architektury představuje testování funkčnosti jedné vygenerované architektury. Pro testování byla vytvořena experimentální architektura, která v sobě kombinuje všechny prvky, které se mohou ve vygenerovaném klasifikátoru vyskytnout. Architekturu tak není možné v reálném systému nasadit, jelikož její celkový klasifikační výsledek je nesmyslný. Architektura však bude demonstrovat veškerou funkčnost, které lze dosáhnout pomocí generátoru klasifikátorů.

V testovací architektuře jsou především zastoupeny všechny typy slabých klasifikátorů. Jsou to LBP, LRP a LR klasifikátory. LBP příznak není možné konfigurovat pomocí generických parametrů, proto bude v architektuře obsažen nejméně krát. Naopak příznaky LRP a LR je možné nakonfigurovat do osmdesáti jednoho stavu pro LRP, respektive do devíti stavů pro LR. Jelikož je uvedený stavový prostor příliš velký, jsou v architektuře zastoupeny jen vybrané konfigurace slabých klasifikátorů.

Pro ověření správné funkčnosti klasifikátoru byl napsán konfigurovatelný klasifikátor v jazyce C++. Klasifikátor napsaný v jazyce C++ má stejné vstupní parametry jako testovací skript (*testbench*) pro VHDL implementaci. Tím je zajištěna snadná změna konfigurace a opětovné ověření výsledků.

V jazyce C++ byly vytvořeny dva odlišné způsoby výpočtu klasifikátoru. Jedna verze je s využitím imperativního programovacího modelu. Druhá verze naopak přesně simuluje každý krok VHDL designu. Je tedy možné simulovat každý krok výpočtu, případně některé kroky vynechat.

Výstup z testovacího programu v C++ a z VHDL testovacího skriptu musí být při správné funkčnosti stejný. To však platí za předpokladu, že z výstupu produkovaného VHDL testovacím skriptem odstraníme data, která souvisí s naplněním pipeline linky v hardwarové realizaci. Pokud by se naplnění hardwarové linky zanedbalo, obsahoval by výstup z hardwarové realizace prefix, který právě reprezentuje naplnění datových linek.

Klasifikátor má několik stupňů pipeline, kterými musí data projít. Počet stupňů je však malý oproti zpoždění, které data nabírají ve výstupní zpožďovací lince (viz 5.5). Zpoždění vygenerované touto linkou je téměř stejné jako velikost detekčního okna (velikost je dána pozicí posledního klasifikátoru). Maximální dosahované zpoždění je například pro detekční okno o velikosti *24 x 24* pixelů a šířku vstupního obrazu 512 pixelů 12283 taktů.

Prozatím opomíjeným faktem, při zpracovávání vstupního obrazu je, že několik posledních zpracovávaných příznaků (pět) pro právě zpracovávaný řádek, není zarovnáno ke konci řádku, ale „přetékají“ do následujícího řádku. Tento fakt je zapříčiněn tím, že se nevyplatí zastavovat linku a raději se nechá vypočítat několik nepotřebných klasifikací, které se v nadřízeném procesu odstraní (například v nadřízeném DSP).

Pro testování byl vybrán silný klasifikátor s následujícími parametry:

Typ	Pozice		RankA	RankB	Typ konvolucí
	X	Y			
LBP	11	6	-	-	72 73 74 75 76 77 78 79 80
LBP	16	0	-	-	36 37 38 39 40 41 42 43 44
LBP	13	4	-	-	0 1 2 6 7 8 12 13 14
LBP	5	6	-	-	54 57 60 55 58 61 56 59 62
LRP	11	4	4	2	54 57 60 55 58 61 56 59 62
LRP	6	7	4	2	0 1 2 6 7 8 12 13 14
LRP	12	12	4	1	36 37 38 39 40 41 42 43 44
LRP	15	17	4	1	0 1 2 6 7 8 12 13 14
LRP	0	0	5	3	54 57 60 55 58 61 56 59 62
LRP	7	12	8	1	72 73 74 75 76 77 78 79 80
LR	17	7	7	-	72 73 74 75 76 77 78 79 80
LR	4	17	6	-	54 57 60 55 58 61 56 59 62
LR	10	8	1	-	36 37 38 39 40 41 42 43 44
LR	11	16	4	-	0 1 2 6 7 8 12 13 14

Tabulka 4: Složení testovacího klasifikátoru.

V tabulce 4 je uvedeno složení testovacího klasifikátoru. První sloupec ukazuje typ použitého slabého klasifikátoru. Pole *pozice* označuje pozici slabého klasifikátoru v rámci detekčního okna. *RanA* a *RankB* jsou hodnoty určené pro výpočet příznaků LRP a LR klasifikátorů. Poslední sloupec s názvem typ konvolucí vyjadřuje, jaké konvoluce se budou v každém slabém klasifikátoru provádět. Konvoluce jsou zadány řadou čísel, kde každé představuje index do tabulky. Typ jednotlivých konvolucí není v tuto chvíli významný. Důležité je ale rozlišení, kolik typů konvolucí je v klasifikátoru obsaženo.

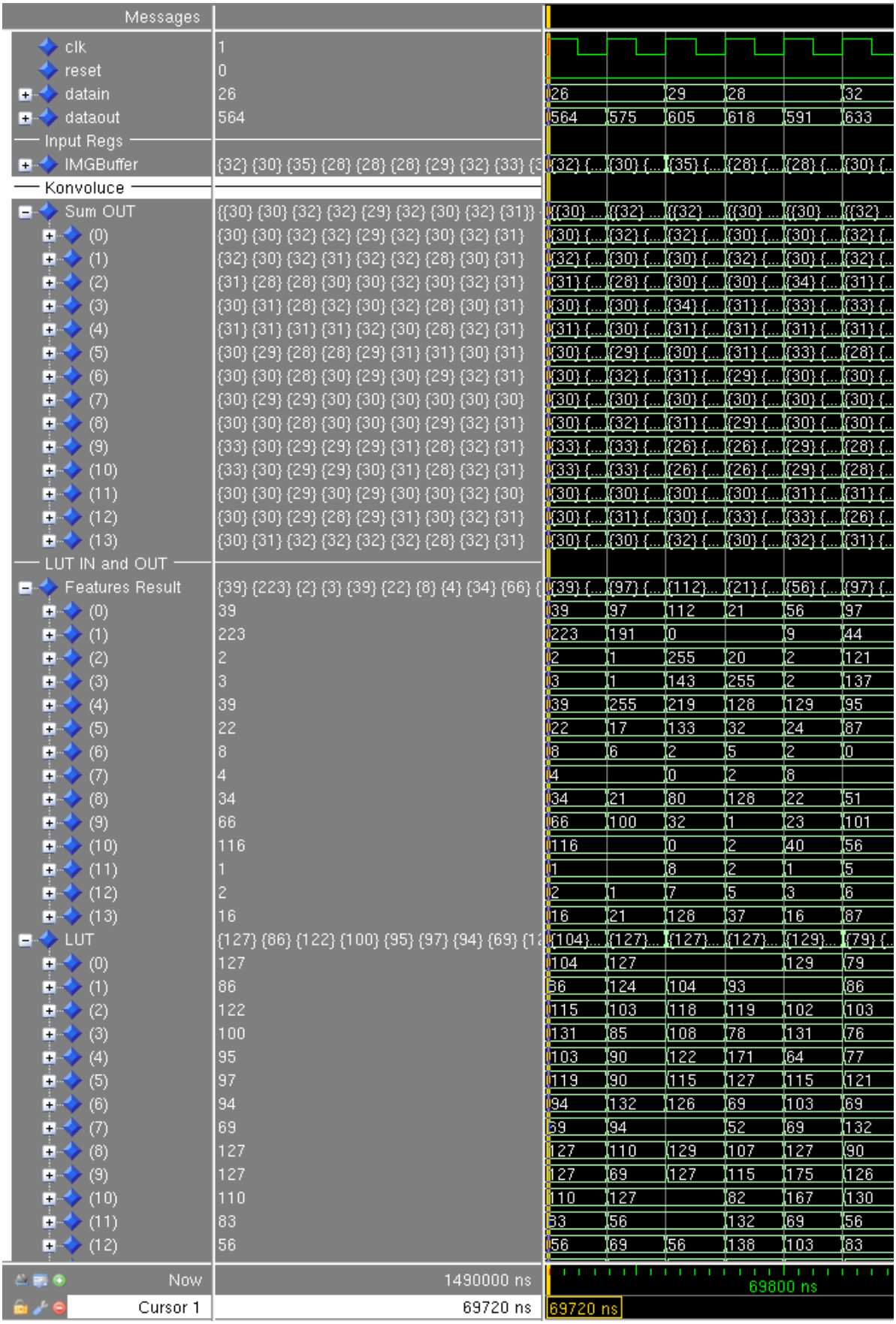
Pro vygenerování požadovaného klasifikátoru musíme spustit generátor, například s následujícími parametry:

```
./vhdlGenClasif -c genVHDLConfig.xml -o DPLBPLRPLR t-AB-LBP-LRP-
LR.xml
```

Kde `genVHDLConfig.xml` je konfigurační soubor, `DPLBPLRPLR` je výstupní adresář a `t-AB-LBP-LRP-LR.xml` je konfigurační soubor s klasifikátorem. Výstupem generátoru je zadaný adresář, ve kterém jsou vygenerovány všechny potřebné zdrojové soubory pro VHDL. Ve vygenerovaném adresáři je již možné provádět simulaci architektury. Pro ověření správnosti výsledků klasifikace je vytvořen program, který lze spustit pomocí příkazu:

```
./tb_clasif genVHDLConfig.xml t-AB-LBP-LRP-LR.xml testData.in
LBPLRPLR.out
```

Program vygeneruje svůj výstup do souboru `LBPLRPLR.out`. Jako vstup si vezme data uložená v souboru `testData.in`. Formát vstupních dat je jak pro *testbench*, který je napsaný ve VHDL, tak i pro testovací program napsaný v C++, totožný. Jelikož ve VHDL není možné zpracovávat binární soubory, je nutné vstupní obrázek nejprve překonvertovat do potřebného formátu. Vstupní soubor je formátován takovým způsobem, že na každém řádku je uvedena dekadická hodnota jednoho pixelu. (Pro přeformátování vstupního obrázku byl vytvořen jednoduchý program `imgConverter`.) Vstupní data musí mít požadovanou délku. Toto omezení je z důvodu, že například šířka obrazu je zakomponována pevně do vygenerovaného VHDL kódu. Pro změnu parametrů je třeba vygenerovat nový návrh.



Obrázek 35: Výstup simulace.

Obrázek 35 ukazuje část výstupu simulace. V horní části obrázku jsou uvedeny globální signály, jako jsou například hodinový signál a signál resetu. Další položky na seznamu jsou vstupní a výstupní signály z klasifikace. Signál `datain` představuje vstup obrazových dat. Signál `dataout` představuje výstup klasifikace. Výstup je v tomto případě za vstupem opožděn o 8719 taktů, které představují zpoždění ve výstupním zpožďovacím bufferu, dalších 6 taktů zpoždění tvoří stupně pipeliningu.

Signál `IMGBuffer` představuje registry pro uložení výřezu obrazu o rozměrech *6 x 6* pixelů. Následující skupina signálů – `Konvoluce` tvoří výstup bloku SUM, hodnoty jednotlivých konvolucí jsou zpožděny oproti výstupu z bloku `IMGBuffer` o jeden takt. Výsledky konvolucí jsou vstupem do příznaků. Výstupy příznaků jsou označeny jako skupina `Features result`. Zpoždění výpočtu příznaků je v této realizaci dva takty (viz 5.3). Výstupy příznaků jsou dle principu architektury přivedeny na vstup vyhledávacích tabulek. Slouží jako adresy do paměťových modů, které realizují vyhledávací tabulky. Výstup vyhledávacích tabulek je přiveden na vstup výstupní zpožďovací linky. Výstup uvedené linky je představován signálem `dataout`. Je však zpožděn o 8720 taktů.

Jednotlivé příznaky a konvoluce jsou uvedeny na obrázku dle pořadí v detekčním okně. Délka jednoho taktu je 20 ns (50 MHz). Pokud vezmeme v úvahu, že máme v čase 69720 lns data na výstupu prvku `IMGBuffer`, tak výsledky konvolucí získáme v čase 69740 ns. V čase 69780 ns získáme výstup z jednotlivých příznaků, které odpovídají vstupu v čase 69720 ns. Výstup vyhledávacích tabulek získáme v čase 69800 ns. Avšak výstup celého systému bychom získali až v logickém čase 244200 ns. Správnost výpočtu příznaků lze jednoduše ověřit pomocí dosazení hodnot konvolucí do vzorců pro výpočet příznaků. Celý proces klasifikace je řízen pomocí *testbench* souboru, který vytvoří výstupní soubor s klasifikacemi. Stejný soubor získáme také pomocí testovacích programů napsaných v jazyce C++.

6.2 Testování architektur

V kapitole testování architektur se budeme zabývat klasifikátory s různým složením, které jsou výsledkem procesu generování. Důležitými parametry, které se budou sledovat v této kapitole, jsou především optimalizace návrhu vzhledem ke spotřebovaným zdrojům na čipu FPGA a rychlost výsledného návrhu.

Pro odhad náročnosti klasifikátorů vzhledem ke zdrojům, bylo vytvořeno několik klasifikátorů, které simulují různé cílové konfigurace. S ohledem na zdroje se bude testovat počet konvolucí, které jsou třeba pro klasifikaci. U LRP a LR příznaků jsou do testovací sady zahrnuty případy, kde je v cílovém návrhu umístěno mnoho různých konfigurací těchto slabých klasifikátorů. Téměř všechny silné klasifikátory mají slabé klasifikátory umístěny na stejných pozicích. Jen jeden Silný klasifikátor má větší velikost detekčního okna (*30 x 30* pixelů). Všechny silné klasifikátory se skládají z dvaceti slabých klasifikátorů. V následující tabulce je uvedena architektura jednotlivých testovaných architektur. Pro syntézu obvodů byl použit nástroj Xilinx ISE WebPack 11.2. Jako cílová architektura, pro kterou se návrh syntetizoval, byl vybrán čip rodiny Spartan3 xc3s1000. Tento čip byl zvolen pro jeho přístupnost v uvedeném syntetizačním nástroji. V reálném nasazení se počítá spíše s čipem rodiny Virtex-II 250.

V tabulce 5 jsou uvedeny výsledky syntézy. Všechny obvody dosahovali téměř stejné rychlosti. Tento jev byl způsoben tím, že nejpomalejší prvek obvodu byl společný pro všechny architektury. Byl jím blok pro výpočet konvolucí. Výpočty příznaků jsou již optimalizovány pro vysokou frekvenci a tak nezpomalují výsledný obvod.

Z výsledků experimentu lze pozorovat, že nejméně zdrojů zabírají klasifikátory založené na LBP příznacích. Je to dáno především tím, že je není možné konfigurovat. Celkový počet

vyhodnocovačů příznaků, který je nutné do výsledné architektury vložit, je závislý jen na počtu různých konvolucí. V nejlepším případě, kdy je klasifikátor tvořen jen LBP příznaky a výpočtem jediné konvoluce zabírá obvod jen 838 *Slices*, což je velmi příznivý výsledek.

Typ příznaků	Počet různých příznaků	Počet konvolucí	Detekční okno	Slices	BRAM	Rychlost
LBP	1	4	24x24	1034	14	130.2 MHz
LBP	1	1	24x24	838	14	132.0 MHz
LRP	20	4	24x24	2627	21	130.2 MHz
LRP	1	4	24x24	1269	14	130.2 MHz
LRP	20	1	24x24	2469	20	132.0 MHz
LR	9	4	24x24	2184	15	130.2 MHz
LR	9	1	24x24	1803	15	132.0 MHz
LR	1	4	24x24	1672	15	130.2 MHz
LBP, LRP	1, 3	4	24x24	1943	18	130.2 MHz
LBP, LRP, LR	1, 3, 3	4	24x24	1828	17	130.2 MHz
LBP	1	4	30x30	1122	19	130.2 MHz

Tabulka 5: Náročnost architektur na zdroje.

Další nespornou výhodou klasifikátorů založených na LBP je počet spotřebovaných *BlockRAM* paměťových modulů. Jejich počet je nejmenší ze všech uvedených řešení, jelikož se podaří velmi dobře optimalizovat jejich využití při tvorbě vyhledávacích tabulek. V uvedeném případě, na druhém řádku v tabulce, jsou využity dvě *BlockRAM* na sto procent a třetí *BlockRAM* je využita jen na dvacet dva procent. Pokud by se zvýšil počet klasifikátorů z dvaceti na dvacet sedm, neznamenalo by to téměř žádné navýšení požadavků na zdroje. Využili by se opět jen tři *BlockRAM* paměti, ale nyní všechny na sto procent.

LRP příznaky poskytují největší množství konfigurací. Proto klasifikátory založené na těchto příznacích zabírají největší množství zdrojů na čipu FPGA. Jsou náročné jednak na počet obsazených *Slices*, ale i na počet použitých *BlockRAM* paměťových modulů, které se spotřebují na vytvoření vyhledávacích tabulek. Využití *BlockRAM* pamětí je velmi malé a v nejhorším případě se může pohybovat okolo dvaceti dvou procent (například pátý řádek v tabulce). Podobně jsou na tom i LR příznaky, které jsou sice méně náročné na zdroje, ale i tak převyšují požadavky LBP příznaků.

Navržená architektura dovoluje kombinovat i více příznaků v jednom klasifikátoru. Ačkoliv toto řešení není v reálných aplikacích příliš využíváno, z hlediska analýzy zdrojů se můžeme podívat na výsledky při využití takové architektury. V tabulce jsou uvedeny tyto konfigurace na devátém a desátém řádku. Dle očekávání jsou nároky na takovou architekturu menší než na architekturu složenou jen z LRP nebo LR příznaků, ale větší než na architekturu slouženou jen z LBP příznaků.

Na posledním řádku tabulky je uveden případ klasifikátoru, který pracuje s větším detekčním oknem. Počet a typy slabých klasifikátorů jsou shodné s klasifikátorem uvedeným na prvním řádku. Slabé klasifikátory se liší jen pozicemi. Proto je celkový počet *Slice* jen nepatrně větší. Avšak bylo použito mnohem více *BlockRAM* paměťových modulů. To je především zapříčiněno větší výstupní zpožďovací linkou, která musí pojmout mnohem více dat.

Nejlepšími klasifikátory, z hlediska využitých zdrojů, jsou tedy klasifikátory složené jen z LBP příznaků. Tyto klasifikátory mají i poměrně dobrou úspěšnost při klasifikaci.

6.2.1 Požadavky na zdroje vybrané architektury

V minulých odstavcích bylo zjištěno, že nejlepší výsledky jsou dosahovány pro architektury založené na LBP příznacích. V testech byly vždy silné klasifikátory složené z dvaceti slabých klasifikátorů a porovnávala se náročnost na zdroje s ohledem na typ použitých slabých klasifikátorů. Nyní vyberme jeden typ slabého klasifikátoru a uděláme odhad náročnosti na zdroje vzhledem k počtu umístěných slabých klasifikátorů. Vybraný slabý klasifikátor bude LBP. Velikost detekčního okna nastavíme na *24 x 24* pixelů. Šířka vstupního obrazu bude 512 pixelů. Dále budeme předpokládat, že na jednom místě v obraze mohou být maximálně jen dva příznaky. Všechny slabé klasifikátory budou pracovat se stejnými vstupními konvolucemi. Počet všech možných různých konvolucí tak bude čtyři. Výsledky byly získány pro FPGA Virtex-II 250.

Počet LBP	Slices	BlockRAM
10	903	14
20	1042	14
30	1201	16
40	1317	17
50	1394	18
60	1451	19
70	1536	20
100	1562	23

Tabulka 6: Spotřebované zdroje vzhledem k počtu slabých klasifikátorů.

Z tabulky 6 lze vidět, že dle očekávání počet spotřebovaných zdrojů narůstá s počtem umístěných slabých klasifikátorů. Nárůst spotřebovaných zdrojů však není velký. Z tabulky lze vidět, že pro deset slabých klasifikátorů potřebujeme 903 *Slices* a pro dvacet klasifikátorů potřebujeme 1042 *Slices*. Pro rozšíření původního klasifikátoru o dalších deset slabých klasifikátorů tak potřebujeme jen asi 140 *Slices*. Lze tak odhadnout, že část architektury, která není závislá na počtu slabých klasifikátorů, zabírá kolem sedmi set *Slices* a dvanácti *BlockRAM* paměťových modulů.

Každé zvětšování počtu klasifikátorů navýší požadavky na zdroje jen nepatrně. S každým nově přidaným klasifikátorem je vždy třeba přidat novou *LUT* a nový prvek do výstupní zpožďovací linky. Není však třeba vždy vytvářet nové konvoluce a vkládat nový blok pro vyhodnocování příznaků. Počet obsazených *BlockRAM* pamětí narůstá dle očekávání jen pozvolna. Jelikož se daří využívat paměti *BlockRAM* na sto procent.

Z tohoto testu vidíme, že jsme schopni na jeden čip FPGA Virtex-II 250 umístit až 50 slabých klasifikátorů. Takový silný klasifikátor již poskytuje velmi dobré výsledky při klasifikaci. Pokud bychom použili větší FPGA čip, mohli bychom na něj umístit až několik stovek slabých klasifikátorů. Takový klasifikátor by poté nepotřeboval ani následný post-processing generovaných dat v počítači.

6.3 Výkonnost architektury

Výkonnost navržené architektury můžeme posuzovat ze dvou hledisek. Prvním hlediskem je porovnání hrubého výpočetního výkonu s dosavadními řešeními. Dalším parametrem může být posouzení efektivnosti výpočtu vzhledem k pořizovací ceně hardwarového zařízení, na kterém

klasifikátor poběží a spotřebě energií, které navržená architektura spotřebuje. Výkonnost hardwarového řešení budeme porovnávat s běžným stolním počítačem.

6.3.1 Výpočetní výkonnost

Výpočetní výkonnost je ukazatel, na který je kladen obvykle největší důraz. Pro posuzování výkonnosti si však nejprve musíme stanovit pravidla, dle kterých budeme pracovat. Jedním možným hlediskem pro posuzování výkonnosti je počet vyhodnocených slabých klasifikátorů (příznaků) za daný čas. Dalším možným hlediskem je počet vyhodnocených detekčních oken za jednotku času. Jelikož je počet vyhodnocení příznaků stejný pro každé detekční okno, jsou obě tyto hlediska rovnocenné. Pro vyhodnocení budu tedy uvažovat druhé uvedené hledisko, a tedy počet vyhodnocených detekčních oken za jednotku času. Časovou jednotkou bude jedna sekunda.

Pro zjištění výkonnosti zavedeme jednotku, která bude vyjadřovat počet vyhodnocení detekčních oken za jednu sekundu - KD/s. U softwarového, ale především i u hardwarového řešení, nebude uvažovat při výpočtu naplnění linek, získání vstupních dat ze souboru a obdobné režijní výpočty. Budeme se zaměřovat především na čas výpočtu příznaků.

Výpočet bude probíhat jednak na běžném počítači s procesorem Intel Pentium M na frekvenci 1,6 GHz. V počítači je nainstalováno dostatečné množství paměti, aby nedocházelo během výpočtu ke čtení dat z pevného disku. Druhá, hardwarová, varianta bude pracovat na FPGA čipu s frekvencí 100 MHz. Obě řešení budou zpracovávat naprosto shodná data a produkovat stejný výstup.

V univerzálním procesoru zabere klasifikace obrazu o rozměrech *512 x 128* pixelů 14,893 sekund. Po přepočítání na zavedenou jednotku dostaneme výkonnost 4202,7 KD/s. Celkový čas výpočtu jednoho snímku je téměř 15 sekund. Takto rychle zařízení není možné uplatnit v aplikaci, která pracuje v reálném čase.

Nyní se podívejme na výkonnost hardwarového řešení ve VHDL. Architektura je navržena tak, že každý hodinový takt produkuje jeden výsledek, který představuje hodnotu klasifikace pro detekční okno. Budeme-li uvažovat jako referenční rychlost FPGA čipu 100 MHz, dostaneme výkonnost 100 000 000 KD/s. V porovnání se softwarovým řešením je 23 798 krát rychlejší. Pokud však použijeme novější čipy FPGA, a klasifikátor složený jen z LBP příznaků, může se pracovní frekvence dostat až na hodnotu 250 MHz. Hardwarové řešení tak bude 59 495 krát výkonnější než běžný stolní počítač. Řešení v software nebylo příliš optimalizováno pro rychlost, ale i kdyby se podařilo zrychlit dvojnásobně a použil by se dvakrát výkonnější procesor, stejně by výkonnost softwarového řešení byla o několik řádů menší

Pokud bychom ukazatele výkonnosti převedli do praxe, například pro zpracování obrazu z kamery, můžeme vypočítat maximální možné rozlišení, které je dané řešení schopno zpracovávat. Předpokládejme konstantní snímkovou rychlost 25 snímků za sekundu. Při klasifikaci pomocí softwarového řešení bychom byli schopni zpracovávat obraz o rozlišení *47 x 29* pixelů. Při zpracování pomocí hardwarové realizace na referenční frekvenci 100 MHz bychom mohli zpracovávat obraz o rozlišení cca *2300 x 1450* pixelů. Takové rozlišení je zcela dostačující pro většinu dostupných obrazových snímačů. Při frekvenci čipu FPGA 250 MHz bychom mohli zpracovávat obraz o rozlišení až *3500 x 2500* pixelů.

6.3.2 Příkon architektury

V této kapitole se podíváme na efektivnost výpočtu, vzhledem ke spotřebované energii na výpočet. Běžný stolní počítač má při zatížení odběr cca 150 W, Výkonnější počítače, uvažuju-li jen jeden procesor, se svou spotřebou pohybují okolo 300 W. (Spotřeba byla odhadnuta na základě spotřeby energie především procesorem a základní deskou, dále pak do celkové sumy byla připočtena

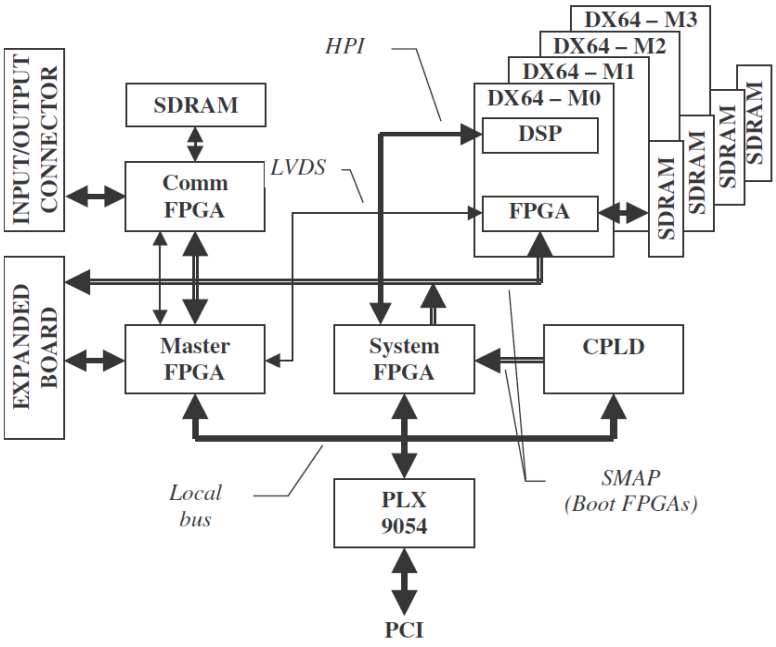
i spotřeba všech nutných komponent pro běh počítače.) Klasifikátor složený z dvaceti LBP, který umístíme na FPGA čip má spotřebu 31,2 mW. Teplota, které čip FPGA může dosahovat je 34,2 stupňů celsia. Čip tak produkuje minimum ztrátového tepla a není jej třeba ani chladit, jako univerzální procesory. Příkonové parametry jsou uvedeny pro FPGA Virtex-II 250.

Efektivnost výpočtu můžeme zobrazit jako počet provedených klasifikací na jeden Watt příkonu. Pro univerzální počítač můžeme za jeden Watt vypočítat cca 10^5 detekčních oken. Při výpočtu pomocí FPGA můžeme spočítat $1{,}1.10^{13}$ detekčních oken. Rozdíl efektivnosti vypočtu je řádově 10^8 ve prospěch FPGA. Pokud by takový detektor měl běžet v reálném čase a nepřetržitě zpracovávat výstup z kamery, jistě se vyplatí implementovat návrh pomocí FPGA technologie. Cena provozu univerzálního počítače je v porovnání s cenou provozu FPGA tak vysoká, že se dražší vývoj zařízení pro FPGA postupně zaplatí na uspořených energiích. Nehledě na to, že realizace v FPGA je mnohem rychlejší.

6.4 Realizace

Tato podkapitola se zabývá zprovozněním klasifikátoru na vybraném čipu FPGA. Jedním z důležitých faktorů pro výběr cílového čipu FPGA je počet zdrojů, které nabízí. Nemalou roli hraje také cena cílového obvodu. Například čipy od firmy Xilinx rodiny Spartan 3 obsahují velký počet *Slices* v poměru k počtu *BlockRAM* pamětí. Jelikož klasifikátor potřebuje hodně pamětí pro vytváření bufferů, nejsou čipy z této rodiny vhodné. Jako vhodné řešení, za stále přijatelnou cenu, se jeví čipy rodiny Virtex-II. Obsahují mnohem více *BlockRAM* pamětí v poměru k obsaženým *Slices*. Například čip Virtex-II 250 obsahuje dvacet čtyři 18kb modulů paměti *BlockRAM*. Současně však obsahuje dostatečný počet 1536 *Slices*. Tento čip se jeví jako optimální volba vzhledem k požadavkům na klasifikátory.

S požadovaným čipem Virtex-II je vytvořen modul s názvem DX-64. Tento modul se skládá z čipu FPGA Virtex-II 250, nebo Virtex-II 1000, a DSP procesoru. Celý modul je možné připojit do architektury s názvem Uni1P.

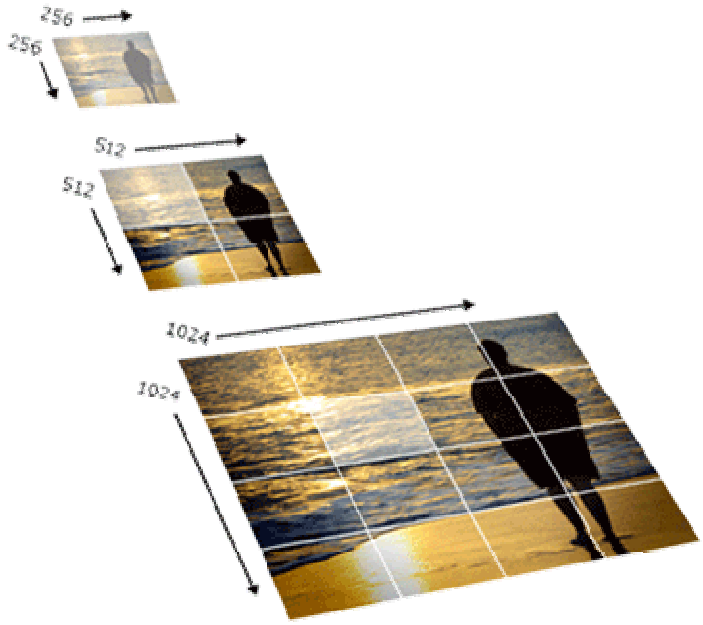


Obrázek 36: Struktura architektury Uni1P [10].

Na obrázku 36 je znázorněna architektura přípravku Uni1P. Tento přípravek umožňuje připojení několika DX64 modulů. Aplikace klasifikátoru však pracuje jen na jednom modulu. Zbývající moduly zůstanou nevyužity. Komunikaci s hostitelským systémem bude obstarávat DPS, umístěné na modulu DX64, přes rozhraní HPI (*Host Port Interface*). Uvedená architektura je použita zejména kvůli její dostupnosti pro testování.

Pro klasifikátor v FPGA je nutné zajistit přísun dat. Jako zdroj dat můžeme uvažovat kameru nebo, pro testovací účely, soubor v hostitelském počítači. Jelikož klasifikátor je naučen klasifikovat objekty jen v určitém rozlišení, musí se vstupní obraz transformovat do několika úrovní rozlišení. Takto transformovaný obraz se nazývá obrazová pyramida. Do klasifikátoru tak postupně vstupuje stejný snímek v několika úrovních rozlišení. Tím je možné detekovat různě velké objekty v původním obraze.

Na obrázku 37 je zobrazena obrazová pyramida. Tato transformace se provádí v DSP, kde je jí možné efektivně implementovat. Obrazová pyramida spočívá v postupném převzorkovávání původního obrazu (zmenšování rozlišení), tak abychom získali několik obrazů s různým rozlišením. Na všech takových obrázcích se následně provede klasifikace.



Obrázek 37: Obrazová pyramida.

O přísun a odebírání dat do a z FPGA se stará DSP. Program, pro DSP, který bude tuto činnost vykonávat, není předmětem této práce a v realizaci bude využit některý z již vypracovaných programů. Výstup klasifikace z FPGA je nutné správně zpracovávat a vynechávat ve výstupu úseky, které jsou způsobené přechodem mezi řádky a zpožděním ve výstupní zpožďovací lince. Jednotlivá zpoždění nejsou konstantní a mění se pro každý klasifikátor. Je tak nutné pro každý klasifikátor vypočítat novou hodnotu zpoždění. Výpočet zpoždění pro inicializaci výstupní zpožďovací linky je uvedeno na straně 34, vzorec 4.1. Při výpočtu každého řádku dochází k přetečení výpočetního okna, o rozměru *6 x 6* pixelů, postupně o jeden, dva až pět pixelů. Na konci každého řádku je tak vygenerováno pět hodnot klasifikace navíc. Tyto zpoždění je nutné v DSP ošetřit. Například pět hodnot na konci řádku je možné zredukovat tak, že budeme výstup klasifikace zpracovávat jako obraz, který je o pět pixelů širší. Tak není nutné data odstraňovat. Jelikož je obraz v této oblasti deformovaný, dá se předpokládat, že v ní nebude žádný pozitivní klasifikační výsledek.

Pro umístění klasifikátoru do modulu DX64 bylo využito implementace rozhraní od pana Antonína Hegara [10]. Pro vložení aplikace klasifikátoru bylo vytvořeno rozhraní, které předává data klasifikátoru. Výsledný design byl úspěšně syntetizován pro platformu Virtex-II 250 a byl úspěšně vytvořen firmware pro tento modul.

7 Závěr

Cílem práce bylo vytvoření klasifikátoru rastrového obrazu v FPGA. Práce se zaměřila především na učící se klasifikátory, které je možné natrénovat na různé druhy klasifikace. Typickým představitelem učícího algoritmu je AdaBoost (viz kapitola 2.6.2). Činnost tohoto klasifikátoru lze rozdělit na dvě fáze. V první fázi se učí předměty klasifikovat a ve druhé fázi předměty jen klasifikuje. Ač je fáze učení v práci zmíněna, není hlavním cílem této práce a tak je zde uvedena jen pro úplnost.

Pozornost se upíná především k druhé fázi klasifikačního procesu. Tou je klasifikace objektů do tříd. Zaměření práce je na klasifikaci rastrového obrazu, proto by bylo zbytečné implementovat složitou fázi učení v FPGA. Na fázi učení máme zpravidla dostatek času, a tak ji není nutné akcelerovat v hardware. Lze ji provádět na běžném osobním počítači. Naučení klasifikátoru se provede jen jednou a poté se již data získaná učením jen používají ke klasifikaci.

Silný klasifikátor, jenž představuje například AdaBoost, je tvořen několika slabými klasifikátory. Vybrání vhodného typu slabého klasifikátoru je pro implementaci v FPGA velmi důležité. V práci jsou představeny klasifikátory založené na Haarových vlnkách, LBP a LRP klasifikátory a některé další vybrané klasifikátory. Z uvedených klasifikátoru jsou vybrány LBP, LRP a LR. Tyto klasifikátory se budou implementovat v cílové architektuře.

Během tvorby práce bylo navrženo několik architektur. Některé byly rozpracovány jen do podoby návrhu a byla na nich provedena základní analýza. Jako výsledná architektura pro realizaci byla zvolena pseudo-paralelní architektura (viz 4.3). Tato architektura, která je založena na principu AdaBoost, se vyznačuje tím, že se nesnaží vytvořit dokonalý klasifikátor, ale snaží se být prvkem, který bude zapojen do většího procesu zpracování dat. Navržený klasifikátor se snaží vyloučit co nejvíce detekčních oken, ve kterých není hledaný předmět. Architektura je navržena tak, že počítá s následným post-procesingem svých výsledků v univerzálním procesoru nebo DSP procesoru. Tento přístup byl zvolen, jelikož není možné za použití běžných univerzálních procesorů provádět v reálném čase klasifikaci obrazu v dostatečném rozlišení obrazu. A zároveň není vhodné provádět celou klasifikaci jen v FPGA čipu. Pokud by se měla celá klasifikace provádět jen v FPGA čipu, musel by se použít velký FPGA čip, což by vedlo na vysoké výrobní náklady.

Výstupem trénovacího procesu může být mnoho různých klasifikátorů. Klasifikátory se mohou lišit použitými slabými klasifikátory, ale i velikostí detekčního okna. Napsat VHDL kód, který by pokrýval všechny možnosti pomocí generických parametrů, není téměř možné. Navíc výstupem trénování jsou vyhledávací tabulky, které se musí do VHDL kódu zapsat. Proto byl vyvinut generátor VHDL kódu, který na základě zadaných parametrů vygeneruje cílovou architekturu.

Jelikož je generátor napsán ve vyšším programovacím jazyce, bylo do něho zakomponováno několik optimalizačních algoritmů. Tyto algoritmy jsou zaměřeny především na optimalizaci využitého místa na čipu FPGA. Největší optimalizace nastává při generování vyhledávacích tabulek, kdy se algoritmus snaží o dosažení co největšího využití *BlockRAM* paměťových modulů.

Škála vygenerovaných klasifikátorů je široká a tak byly provedeny testy, které klasifikátory jsou nejvhodnější pro implementaci v FPGA, z hlediska rychlosti a místa spotřebovaného na čipu FPGA. Jako nejlepší volba se ukázaly klasifikátory založené na LBP příznacích. Tyto klasifikátory, díky své nemožnosti konfigurovat je, zabírají nejméně místa na čipu a současně jsou nejrychlejší (viz 6.2).

V kapitole 6.3 je uvedeno srovnání výkonnosti hardwarového řešení a softwarového řešení. Dle předpokladů je hardwarové řešení mnohonásobně rychlejší než softwarové řešení. Zajímavým experimentem v kapitole 6.3.2 je přepočet výkonnosti vzhledem ke spotřebovaným energiím. Zde se

projevil velmi vysoký rozdíl mezi výpočtem v FPGA a výpočtem na univerzálním procesoru. Na FPGA spotřebujeme pro výpočet stejného množství dat sto milionkrát méně energie.

Na závěr se práce zabývá reálnou implementací klasifikátoru ve vybraném čipu. Jako vhodný kandidát byl vybrán čip Virtex-II 250, který obsahuje za přijatelnou cenu dostatečný počet pamětí *BlockRAM* a *Slices*. S čipem Virtex-II 250 je vyroben modul DX64. Společně s uvedeným FPGA čipem je na modulu ještě DSP procesor. Uvedený DSP se stará o komunikaci s nadřízeným systémem a o přípravu dat do FPGA.

Projekt byl úspěšně navázán na probíhající projekty na Fakultě Informačních Technologií. Během činnosti spolupracovali na projektu především Pavel Zemčík a Roman Juránek. Roman Juránek měl za úkol především natrénování klasifikátorů a jejich dodání pro účely projektu. Během spolupráce byl podán článek na konferenci ACIVS[16] 2010 (*Automatic Synthesis of AdaBoost Classifiers* – Automatická syntéza AdaBoost klasifikátorů). Předpokládá se, že se na projektu bude nadále pracovat a budou vydány další publikace.

7.1 Budoucí vývoj

Další rozvoj práce se předpokládá především v dokončení implementace pro modul DX64 a jeho případné napojení na videokameru. V současném řešení jsou použity lineární obrazové filtry pro výpočet konvolucí. Současné filtry se snaží popisovat jen texturní vlastnosti obrazu. V dalším vývoji lze očekávat například využití morfologických filtrů. Případně filtrů, které mohou obraz vhodně transformovat, za účelem zvýšení úspěšnosti klasifikace. Takovým filtrem může být například detektor hran, který v obraze zvýrazní hrany.

Během implementace klasifikátoru ve VHDL bylo zjištěno několik skutečností, jejichž aplikací lze zvýšit úspěšnost klasifikace za minimální nebo žádné navýšení zdrojů na čipu FPGA. Tyto poznatky je vhodné zahrnout zpětně do trénovacího procesu, který by tak mohl generovat již částečně optimalizované architektury. Jedná se především o počet stejných slabých klasifikátorů na čipu FPGA (klasifikátory se liší jen svou pozicí a obsahem vyhledávací tabulky). Optimalizační algoritmus pro generování FPGA architektury bude produkovat nejlepší výsledky, pokud bude počet „shodných“ klasifikátorů 9 nebo 4 (viz 5.4).

Další doposud neimplementovanou možností je kombinace několika druhů slabých klasifikátorů v jednom silném klasifikátoru. Dosavadní trénovací procesy dokážou pracovat jen s jedním typem slabého klasifikátoru. Architektura však dovoluje využití všech typů slabých klasifikátorů najednou. V této oblasti je třeba udělat výzkum a zjistit, zda takové klasifikátory budou přínosem.

[16] ACIVS – Advanced Concepts for Intelligent Vision Systems – Pokročilé metody pro inteligentní video systémy.

Seznam obrázků

Literatura

[1] WWW stránky. Virtex-6 Family Overview[online]. http://www.xilinx.com/ (prosinec 2009).

[2] WWW stránky. AVNET [online]. http://avnetexpress.avnet.com/ (duben 2010).

[3] Virtex-II Pro and Virtex-II Pro X FPGA User Guide [online]. http://www.xilinx.com/support/documentation/user_guides/ug012.pdf (duben 2010).

[4] Cho, J., Mirzaei, S., Oberg, J., Kastner, R.: Fpga-based face detection systém using Haarclassifiers. In *FPGA '09: Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*, New York, NY, USA: ACM, 2009, ISBN 978-1-60558-410-2, s. 103-112.

[5] Choi, W., Tse, S., Wong, K., Lam, K.: Simplified Gabor wavelets for human face recognition. *Pattern Recognition*, ročník 41, č. 3, 2008: s. 1186-1199, ISSN 0031-3203, part Special issue: Feature Generation and Machine Learning for Robust Multimodal Biometrics.

[6] Freund. Y., Schapire, R.: A short introduction to boosting. *Soc. for Artif*, ročník 14, č. 5, 1999: s. 771-780.

[7] Freund, Y., Schapire, R. E.: A decision-theoretic generalization of on-line learning and an application to boosting. In *EuroCOLT '95: Proceedings of the Second European conference on Computational Learning Theory*, London, UK: Springer-Verlag, 1995, ISBN 3-540-59119-2, s. 23-37.

[8] Ftáčnik, M.: Rozpoznávanie obrazcov [online]. http://www.sccg.sk/~ftacnik/ (prosinec 2009).

[9] Haar, A.*: Zur Theorie der orthogonalen Funktionensysteme*, chapter Mathematische Annalen. 1910, s. 331-371.

[10] Hegar, A.: Uni1P – Registry. Camea s.r.o., 2006.

[11] Herout, A., Zemčík, P., Hradiš, M., Juránek, R., Havel, J., Jošth, R., Žádník, M.: *Low-Level Image Features for Real-Time Object Detection*. IN-TECH Education and Publishing, 2010, ISBN 978-953-7619-90-9, str. 25.

[12] Herout, A., Zemčík, P., Juránek, R., Hradiš, M.: Implementation of the "Local Rank Differences" Image Feature Using SIMD Instructions of CPU. In *Proceedings of Sixth Indian Conference on Computer Vision, Graphics and Image Processing*, IEEE Computer Society, 2008, ISBN 978-0-7695-3476-3, str. 9.

[13] Herout, A.,Jošth, R., Zemčík, P., Hradiš, M. : GP-GPU Implementation of the "Local Rank Differences" Image Feature. In *Proceedings of International Conference on Computer Vision and Graphics 2008, Lecture Notes in Computer Science, Springer Verlag, 2008*, ISBN 978-3-642-02344-6, s. 1-11.

[14] Hradiš, M.: *AdaBoost v počítačovém vidění*. Diplomová práce, Fakulta informačních technologií, VUT v Brně, 2007.

[15] Hradiš, M., Herout, A., Zemčík, P.: Local Rank Patterns - Novel Features for Rapid Object Detection. In *Proceedings of International Conference on Computer Vision and Graphics 2008*, číslo 12 in Lecture Notes in Computer Science, Springer Verlag, 2008, ISSN 0302-9743, s. 1-12.

[16] Juránek, R.: *Rozpoznávání vzorů v obraze pomocí klasifikátorů*. Diplomová práce, Fakulta informačních technologií, VUT v Brně, 2007.

[17] Kostin, A.: A simple and fast multi-class piecewise linear pattern classifier. *Pattern Recognition*, ročník 39, č. 11, 2006: s. 1949-1962, ISSN 0031-3203.

[18] Krontorád, J.: *Implementace algoritmu SVM v FPGA*. Diplomová práce, VUT v Brně, 2009.

[19] Lee, T.: Image representation using 2D Gabor wavelets. ročník 18, č. 10, 1996: s. 959-971.

[20] Li, X., Kin-Man, L., Shen, L., Zhou, J.: Face detection using simplified Gabor features and hierarchical regions in a cascade of classifiers. *Pattern Recognition Letters*, ročník 30, č. 8, 2009: s. 717-728, ISSN 0167-8655.

[21] Liao, S., Zhu, X., Lei, Z., et al.: Learning Multi-scale Block Local Binary Patterns for Face Recognition. Chinese Academy of Sciences, China, 2007, s. 828-837.

[22] Mäenpää, T.: *The Local Binary Pattern approach to texture analysis - extensions and applications*. Dizertační práce, Faculty of Technology, University of Oulu, 2003.

[23] Maršík, L.: *Zpracování obrazu v FPGA*. Bakalářská práce, VUT v Brně, 2008.

[24] Nanni, L., Lumini, A.: Local binary patterns for a hybrid fingerprint matcher. *Pattern Recognition*, ročník 41, č. 11, 2008: s. 3461-3466, ISSN 0031-3203.

[25] Ojala, T., Pietikäinen, M., Harwood, D.: A comparative study of texture measures with classification based on featured distributions. *Pattern Recognition*, ročník 29, č. 1, 1996: s. 51-59, ISSN 0031-3203.

[26] Pietikäinen, M.: Image Analysis with Local Binary Patterns. In *Image Analysis*, University of Oulu, Finland, 2005, ISBN 978-3-540-26320-3, s. 115-118.

[27] Polok, L., Herout, A., Zemčík, P., Hradiš, M., Juránek, R., Jošth, R.: "Local Rank Differences" Image Feature Implemented on GPU. In *ACIVS '08: Proceedings of the 10th International Conference on Advanced Concepts for Intelligent Vision Systems*, Berlin, Heidelberg: Springer-Verlag, 2008, ISBN 978-3-540-88457-6, s. 170-181.

[28] Skurichina, M.*: Stabilizing Weak Classifiers*. Diplomová práce, Technische Universiteit Delft, 2001.

[29] Sochman, J., Matas, J.: WaldBoost - Learning for Time Constrained Sequential Detection. In *CVPR '05: Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Volume 2*, Washington, DC, USA: IEEE Computer Society, 2005, ISBN 0-7695-2372-2, s. 150-156.

[30] Tajeripour, F., Kabir, E. and Sheikhi, A.: Fabric defect detection using modified local binary patterns. *EURASIP J. Adv. Signal Process*, ročník 2008, 2008: s. 1-12, ISSN 1110-8657.

[31] Theodoridis, S., Koutroumbas, K.: *Pattern recognition*. Academic press, druhé vydání, 2003, ISBN 0-12-685875-6.

[32] Viola, P. and Jones, M.: Rapid object detection using a boosted cascade of simple features. In *CVPR*, 2001.

[33] Viola, P., Jones, M.: Robust Real Time Object Detection. *SCTV*, 2001: s. 1949-1962, Vancouver, Canada.

[34] Viola, P., Jones, M.: Robust Real Time Object Detection. In *SCTV*, Vancouver, Canada, 2001.

[35] Wilson, P. I., Fernandez, J.: Facial feature detection using Haar classifiers. *J. Comput. Small Coll.*, ročník 21, č. 4, 2006: s. 127-133, ISSN 1937-4771.

[36] Wrhel, V.: *Aplikace algoritmu AdaBoost*. Diplomová práce, VUT v Brně, 2008.

[37] Zbořil, F. V.: Opora k předmětu Základy umělé inteligence[online]. https://www.fit.vutbr.cz/study/courses/IZU/private/ (prosinec 2009).

[38] Zemčík, P.: Hardware acceleration of graphics and imaging algorithms using FPGAs. In *SCCG'02: Proceedings of the 18th spring conference on Computer graphics*, New York, NY, USA: ACM, 2002, ISBN 1-58113-608-0, s. 25-32.

[39] Zemčík, P., Žádnik, M.: AdaBoost Engine. In *Field Programmable Logic and Applications*, Aug. 2007, s. 656-660.

[40] Zhang, L., Chu, R., Xiang, S., et al.: Face Detection Based on Multi-Block LBP Representation. In *Advances in Biometrics, Chinese Academy of Sciences*, 2007, ISBN 978-3-540-74548-8, s. 11-18

[41] Španěl, M., Beran, V.: Obrazové segmentační techniky[online]. http://www.fit.vutbr.cz/~spanel/segmentace/ (prosinec 2009), 2006.

Seznam příloh

Příloha 1: CD se zdrojovými kódy.

A Práce s generátorem VHDL kódu

Program pro generování VHDL kódu je umístěn na přiloženém CD ve složce `genVHDL`. Překlad programu se provede pomocí makra `Makefile`. Při překladu se vygenerují následující soubory:

- `vhdlGenClasif` – program pro generování VHDL kódu.
- `tb_clasif` – softwarová implementace klasifikátoru.
- `imgConverter` – program na převod obrázků do požadovaného formátu pro vstup do klasifikace.

Program `vhdlGenClasif` vyžaduje pro svůj běh několik vstupních parametrů:

- `-c soubor` – název konfiguračního souboru
- `-o adresář` – název výstupního adresáře pro generování
- `soubor` – soubor s popisem klasifikátoru

Příklad spuštění generátoru:

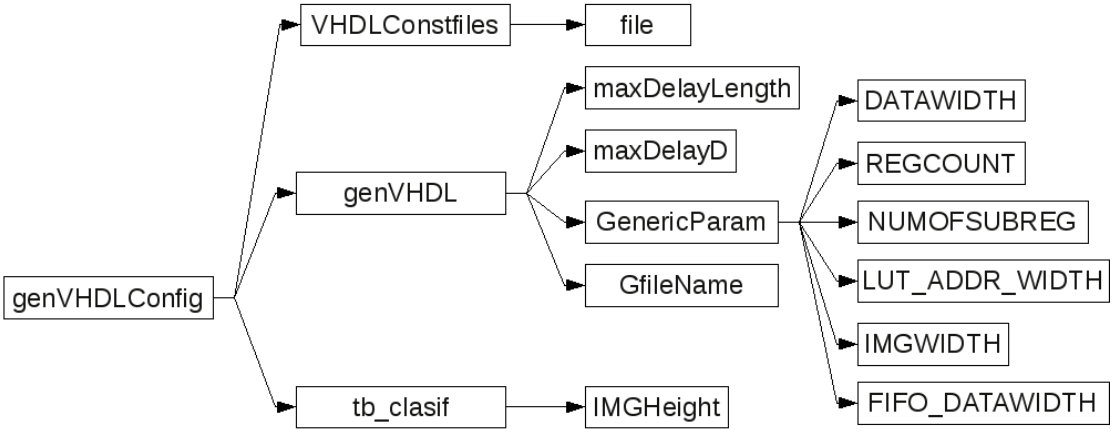
```
./vhdlGenClasif -c genVHDLConfig.xml -o TestDIR clasifier.xml
```

Uvedený příklad očekává konfigurační soubor s názvem `genVHDLConfig.xml`. Svůj výstup vygeneruje do adresáře `TestDIR`. Pokud bude adresář před generováním kódu existovat, jeho obsah se smaže a vytvoří se nový. Popis klasifikátoru je očekáván v souboru `clasifier.xml`.

Vygenerovaný adresář bude obsahovat zdrojové kódy jazyka VHDL. Program pro generování VHDL kódu není přenositelný na úrovni binárního souboru. Pro svůj běh vyžaduje složku, ve které jsou umístěny šablony pro tvorbu VHDL kódu, a složku, která obsahuje neměnné VHDL soubory.

B Konfigurační soubor generátoru VHDL

Soubor slouží pro nastavení parametrů pro generování. Obsahuje odkazy na neměnné VHDL soubory, generické parametry pro VHDL, jména výstupních souborů a další nastavení. Tento soubor je stejný pro programy `vhdlGenClasif` a `tb_clasif`. Formát souboru je XML.



Obrázek 38: Struktura konfiguračního souboru.

C XML popis silného klasifikátoru

Soubor s popisem XML klasifikátoru je výsledkem trénovacího procesu. Jelikož proces trénování není obsahem této práce, není možné ovlivnit strukturu souboru a využívá se již zavedeného formátu. Vstupní XML soubor může nabývat různých podob a s postupujícím časem se může měnit. Pro proces generování VHDL kódu jsou důležité jen jeho některé části. Jelikož je soubor v XML formátu, můžeme tak lehce zpracovat jen vybrané části. Element XML souboru, který popisuje klasifikátory se nazývá `HistogramWeakHypothesis`.

Může vypadat následovně:

```
<HistogramWeakHypothesis predictionValues="-1.31817 … 0.109416 ">
     <LBPFeature   positionX="11"   positionY="6"   blockWidth="2"
     blockHeight="2" selection="72 73 74 75 76 77 78 79 80 " />
</HistogramWeakHypothesis>
```

Atribut `predictionValues` představuje obsah vyhledávacích tabulek. Celkem musí obsahovat 256 hodnot v případě LBP. Hodnoty jsou uváděny ve formátu *float*. Avšak počet všech možných hodnot, které se mohou vyskytnout, je jen 256. Rozsah hodnot je od -3,0 do 3,0. Hodnoty tak lze lehce normalizovat do dekadického tvaru o rozsahu čísel 0 až 255.

Element `<LBPFeature>` určuje typ slabého klasifikátoru. Namísto tohoto elementu mohou být uvedeny definice LRP a LR klasifikátorů. Parametry `positionX` a `positionY` určují pozici příznaku v rámci detekčního okna. Parametr `selection` popisuje typy konvolucí, které se použijí pro výpočet. Redundantní položkou jsou `blockWidth` a `blockHeight`. Tyto parametry též popisují typ konvoluce. Pro samotné generování nejsou potřeba.

V klasifikátorech LRP a LR se mohou objevit ještě parametry `blockA` a `blockB`, které určují pozici bloku pro porovnávání dle principu LRP nebo LR. Pro definici LRP se použije element a názvem `<LRP>`, pro LR element s názvem `<LR>`.

D Popis konvolucí pro slabé klasifikátory

Pro výpočet konvolucí byl zaveden speciální formát. Každý klasifikátor má ve své definici uveden parametr `selection`. Parametr je indexem do pole definovaného v hlavičkových souborech jazyka C v programu. Formát hlavičkového souboru je přebrán z trénovacího procesu, a proto jej není možné ovlivnit. V hlavičkovém souboru je definována každá konvoluce zvláště jako součet pixelů s následným dělením. Tento soubor se nedoporučuje měnit bez hlubší znalosti procesu trénování klasifikátorů. Jeho změna, bez nového natrénování klasifikátoru, by vedla chybné funkčnosti klasifikace. Soubor je při překladu přikompilován k programu.

E Obsah CD

```
CD +
   +-examples/ +                Příklady
   |           +-clasifiers     Ukázky konfiguračních souborů
   |           |                klasifikátorů
   |           +-config         Ukázkový konfigurační soubor
   |           +-testData       Vstupní data pro testování
   |
   +-doc/                       Text diplomové práce
   |
   +-DX64Template/              Šablona pro vytvoření realizace na
   |                            DX64 modulu
   |
   +-test/ +                    Příklad textu aplikace
   |       +-LBPs20             Konfigurace s LBP, 20 klasifikátorů
   |       +-LBP_LRP_LR         Konfigurace s LBP, LRP, LR
   |       +-DX64LBPs20         Ukázka implementace pro DX64
   +-genVHDL/ +
              +-src/            Zdrojové soubory
              +-Makefile        Makro pro překlad
              +-vhdlConstFiles/ Neměnné VHDL soubory
              +-vhdlTemplates/  Šablony pro generování VHDL kódu
```

Ve výčtu jsou uvedeny jen vybrané soubory a adresáře.